NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE IMPACT ON QUALITY OF SERVICE WHEN USING SECURITY-ENABLING FILTERS TO PROVIDE FOR THE SECURITY OF RUN-TIME VIRTUAL ENVIRONMENTS

by

Ernesto Jose Salles

September 2002

Thesis Advisor:	Bret Michael
Thesis Co-Advisor:	Michael Capps
Second Reader:	Don McGregor

This thesis done in cooperation with the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2002	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: The Impact on Quality of Service When Using Security-Enabling Filters to Provide for The Security of Run-Time Extensible Virtual Environments			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ernesto Jose Salles			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

The Naval Postgraduate School is developing NPSNET-V, a Run-Time Extensible Virtual Environment (RTEVE) framework. RTEVEs differ from traditional VEs in that applications within the environment can both discover and use new object types and behaviors at runtime. As the use of this technology has become more valuable to organizations, the need for adequate security has arisen, particularly for sensitive military and commercial applications. The level of security measures employed by these applications must be weighed against their impact on Quality of Service (QOS).

To address RTEVE security issues, we developed a taxonomy identifying twenty-five information assurance (IA) areas within RTEVEs. We then designed and implemented a Security Management System for NPSNET-V (NSMS) that provided security through the use of three communications filters that provide for encryption, sequencing verification, and integrity. This design addressed four of the twenty-five areas identified in the taxonomy: component authentication; and communications confidentiality, integrity, and authentication.

Analysis of the encryption, sequencing, and integrity filters indicates that their use introduces a negligible delay of 0.111 milliseconds for a 156 byte data packet, at the cost in packet size increase of 41 bytes; this indicates the technical feasibility of RTEVE data packet security at minimal cost to QOS.

14. SUBJECT TERMS: Virtual Environment, Virtual Reality, Collaborative Environment, Run-time Extensible Virtual Environment, Security, Quality of Service, Cryptography, Integrity, Encryption			**15. NUMBER OF PAGES** 151
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

THE IMPACT ON QUALITY OF SERVICE WHEN USING SECURITY-ENABLING FILTERS TO PROVIDE FOR THE SECURITY OF RUN-TIME EXTENSIBLE VIRTUAL ENVIRONMENTS

Ernesto J. Salles
Lieutenant Commander, United States Navy
B.S., Tulane University, 1992

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
September 2002**

Author: Ernesto J. Salles

Approved by: J. Bret Michael
Thesis Advisor

Michael Capps
Co-Advisor

Don McGregor
Second Reader

Rudy Darken
Chair, MOVES Academic Committee

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Naval Postgraduate School is developing NPSNET-V, a Run-Time Extensible Virtual Environment (RTEVE) framework. RTEVEs differ from traditional VEs in that applications within the environment can both discover and use new object types and behaviors at runtime. As the use of this technology has become more valuable to organizations, the need for adequate security has arisen, particularly for sensitive military and commercial applications. The level of security measures employed by these applications must be weighed against their impact on Quality of Service (QOS).

To address RTEVE security issues, we developed a taxonomy identifying twenty-five information assurance (IA) areas within RTEVEs. We then designed and implemented a Security Management System for NPSNET-V (NSMS) that provided security through the use of three communications filters that provide for encryption, sequencing verification, and integrity. This design addressed four of the twenty-five areas identified in the taxonomy: component authentication; and communications confidentiality, integrity, and authentication.

Analysis of the encryption, sequencing, and integrity filters indicates that their use introduces a negligible delay of 0.111 milliseconds for a 156 byte data packet, at the cost in packet size increase of 41 bytes; this indicates the technical feasibility of RTEVE data packet security at minimal cost to QOS.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
CVE	Collaborative Virtual Environment
DES	Data Encryption Standard
DESede	Triple DES or Multiple DES
DOS	Denial-of-Service
DSM	Distributed Security Management system
ESPDU	Entity State Packet Data Unit
IA	Information Assurance
I&A	Identification and Authentication
IP	Internet Protocol
IDS	Intrusion Detection System
Jar	Java archive
JCE	Java Cryptographic Extension
JSSE	Java Secure Socket Extension
LDAP	Light-weight Directory Access Protocol
MLS	Multi-level Security
MUD	Multi-User Dungeon
NPSNET-V	Naval Postgraduate School Network – version V
NTP	Network Time Protocol
PKI	Public Key Infrastructure
RTEVE	Runtime Extensible Virtual Environment
SHA	Secure Hash Algorithm
SIMNET	Simulator Networking
SSL	Secure Socket Layer
TCP	Transfer Control Protocol
URL	Uniform Resource Locator
VE	Virtual Environment
VO	Virtual Organization
XML	Extensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

My many thanks and appreciations go to Bret Michael, Michael Capps, Don McGregor, and Andrzej Kapolka. Their guidance and assistance is what has allowed this work to come to fruition. Additionally, their work with me as coauthors in [Salles02] provided a foundation upon which to explore the use of different types of filters for addressing security concerns associated with RTEVEs.

My deep love and thanks to my parents and siblings, who imbued in me the obssessiveness and perseverance that keeps me going, even when times are rough.

To my friends and dance partners at the Blue-Fin, the stress relief provided by the Swing dancing on Tuesday nights was always a welcome distraction from the thesis work.

And finally to Pandora, my puppy-cat, whose doggish antics constantly amused and confounded me, and reminded me that all is not as you expect it to be.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

The term virtual environment (VE) can be used to describe many different types of systems. The scope of this work is centered in the context of networked visual VEs, in which participants are interacting with other participants in a 'real-time' manner. Participants are depicted visually through the use of an image/model (commonly referred to as an 'avatar') in a two- or three-dimensional manner. These are different from non-visual collaborative virtual environments (CVEs) and virtual organizations (VOs) that share resources and data strictly for computational reasons, and have no requirement for real-time visual depictions of the interactions.

A special type of networked visual virtual environment is the Run-Time Extensible Virtual Environments (RTEVEs). Traditional VEs can only operate with objects and behaviors that are present when the VE was started; if any kind of new object type needs to be added to the VE, the VE would need to be halted, the new object-type inserted into the database, and then the VE restarted. RTEVEs are useful for systems that require continuous operation coupled with the ability to add new objects, behavior, and functionality at runtime. This also requires that not only data also code modules be passed over the network.

Networked visual VEs must have significant minimum latency requirements in order to retain a sense of presence. The VE must respond and interact with other participants within the expected human reaction times. [Singhal99]. Given these limitations, security, which unavoidably adds to latency, has usually been sacrificed or ignored. Contributing to this tendency has been a lack of requirements for security in VEs designed for academic research. Military VEs, such as SIMNET (Simulator Networking) [Singhal99], were one of the few VE areas that had a requirement for security. This requirement was typically met by three techniques: physical security for the hosts and local area network; dedicated wide area networks, not accessible from public networks; and network link layer encryption provided by dedicated hardware for wide area network communications. With the dramatic increase in computing and network speed over the

past few years, networked VEs have become more widespread. Commodity, low-cost desktop PCs networked over high speed links can be used as fully participating hosts in visual networked VEs. And this has allowed the widespread deployment of VE applications on public networks that are not secured against attack. This new environment cannot use the same measures that the military used to secure their VEs. LANs cannot always be physically secured against all intruders in a shared environment, and WAN connections must be shared with other users. Link layer security is sometimes impractical for certain types of VE communications. As security-sensitive VE applications are developed for networked VEs, these security issues must be addressed. The increase in computing power has also allowed mechanisms that were previously considered impractical to be deployed.

NPSNET-V was developed at the Naval Postgraduate School and is a framework for the development and research of RTEVE applications. A main design goal of NPSNET-V is to be flexible and deployable on public networks. Therefore, security of the network cannot be assumed. Consequently, any desired level of security must reside within the application itself.

The motivation for this thesis is twofold. The first goal is to develop a taxonomy that identifies the areas of security concern within the domain of RTEVEs. The second goal is to provide the foundation for a security capability within NPSNET-V, and identify its impact on Quality of Service (QOS).

B. THESIS STATEMENT

The use of security-enabling filters that provide for the encryption, sequencing, and integrity of data packets within an RTEVE are effective at addressing relevant RTEVE Information Assurance (IA) concerns with minimal impact on the QOS areas of delay and bandwidth.

C. METHODOLOGY

This section describes the overall methodology that was followed in developing a taxonomy describing twenty-five security related areas within RTEVEs, and the subsequent design and analysis of the basic security management system that was designed for NPSNET-V. Since NPSNET-V initially had no security capabilities, and a VE's main characteristic is the multitude of packets that are exchanged when updating

state information, the design for this base system revolved around the data transmission flows for entity update information. And, in keeping with the QOS concerns that are central to networked systems, various analyses were performed to determine the impact of the added security mechanisms.

1. Review of Virtual Environments

A look into the security of RTEVEs required first an understanding of the typical VE architecture, the desired characteristics of effective VEs, and the QOS issues to be considered when determining what security measures to use. A comprehensive look at VEs can be found in [Singhal99].

2. Analysis of RTEVE Architecture, Case Study of NPSNET-V

In order to identify pertinent security-related concerns and areas within the realm of RTEVEs, an actual RTEVE architecture, NPSNET-V was reviewed and analyzed. Analysis of its structure and design allowed for the breakdown of the system into 5 identified functional areas.

3. Review of Related Security Research

Much research has been performed in the topic of information assurance (IA). However, security research specific to VEs has been somewhat limited. A thorough review of research applicable to VE security was undertaken, with the intent of assisting in the development of the taxonomy, and providing information about the policy and mechanisms that can be applied to address the identified RTEVE security areas.

4. Develop A Taxonomy of RTEVE Security Concerns

The analysis of NPSNET-V and the study of security technology research were then used to develop a taxonomy describing twenty-five RTEVE security areas. The developed taxonomy is further explained through the use of scenarios that assist in understanding the concepts. Further, the research was used to identify security measures applicable in addressing individual and combined areas of the taxonomy.

5. Design and Implementation of the NSMS

Having identified the need to address security in the entity state data packet transmission infrastructure, and having the knowledge of appropriate mechanism to employ, the design and implementation of the SMS was the next logical step. An appropriate design was developed and implementation executed. Since NPSNET-V is a

Java-based framework, the Java Security Application Programming Interface (API) was identified as the appropriate basis of providing security functionality for the NSMS. The design includes a server-client authentication process with symmetric key-distribution for encryption of communication links. Functionality of the system provides for communication enciphering and deciphering, packet data integrity verifications, and packet sequencing operations.

Four of the twenty-five identified areas are addressed by the designed system. These areas are:

- Component authentication: addressed by the authentication of one particular type of component, the *StandardSecurityManager*, via digital certificates.
- Communication authentication: addressed by applications having knowledge of a secret key.
- Communication integrity: addressed through the use of integrity verifying message digests.
- Communication confidentiality: addressed through the use of symmetric key encryption.

6. Analysis of NSMS Capabilities on QOS

Once the implementation was functional and in place, we then studied the impact of the functional areas of the NSMS on QOS, primarily bandwidth and delay.

D. RESULTS

This work has resulted in a greater understanding of the domain of security within RTEVEs, as evinced by the developed matrix of twenty-five RTEVE security areas. It must be remembered, that this taxonomy only covers the RTEVE application realm itself, and assumes other areas of information assurance are handled by the systems on which RTEVEs are executed. Mechanisms exist that can be used to address these areas, but these can come at the cost of QOS which is even especially vital for VE applications.

The development of the NPSNET-V Security Management System (NSMS) indicates that it is feasible to incorporate entity state update data security within RTEVEs.

Additionally, the security mechanisms that can be applied to the data packet exchange infrastructure can be effective with minimal impact to QOS issues.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND AND RELATED WORK

This chapter provides the reader with an understanding of why security is relevant to VEs, an overview of VEs, as well as a discussion on RTEVEs using NPSNET-V as a case study. An overview of current security work relevant to VEs is presented, followed by an overview on technologies used in the implementation of the NSMS.

A. INTRODUCTION

Historically, the Department of Defense was the primary developer of networked VEs. Its early work with training simulators, such as SIMNET (SIMulator NETworking) paved the way for today's VEs [Singhal99]. These early VE efforts required large amounts of computing resources and specialized systems, and thus security of those VEs rested primarily on the fact that the networks were not directly accessible via the Internet.

With the proliferation of PC workstations, and their increasing computing power, distributed, real-time VEs are becoming more commonplace and are used for a multitude of applications ranging from training to manufacturing and gaming. Their architectures have traditionally been driven by QOS considerations, with little concern paid to security issues. However, the new application domains have significant security requirements, explicitly recognized or not, that may involve proprietary or valuable information.

B. SECURITY AND VIRTUAL ENVIRONMENTS

This section briefly discusses the need for security within VEs. It also discusses the idea of a continuum of levels of support for IA, and how the unique characteristics of various VE applications may require specialized security measures.

1. Why Security is Necessary

More and more organizations have realized what the on-line gaming industry and military have known for years: VEs are more revenue generating applications, cost cutting team training and development tools, and safe training environments. As the value and potential of VEs become better understood, they will likely be utilized for an ever greater diversity of applications. Many of these VEs will be used in sensitive contexts that make them targets of malicious entities. Consider the following examples:

Example #1: Manufacturers may use VEs for virtual prototyping, allowing engineers at several locations to make design decisions in a collaborative environment in which they are all virtually present and observe proprietary information. A competitor, engaging in industrial espionage, might be able to view the proprietary information and use it to the their advantage in the marketplace, or they may modify or destroy the information to mislead or disrupt the target corporation.

Example #2: A military commander may utilize a VE to visualize the battlefield and all pertinent intelligence information during a conflict. An adversary that is able to exploit weaknesses in the VE could inject false information, causing the commander to make misinformed decisions, such as mistakenly sending troops into an orchestrated ambush.

2. Varying Levels of Security

As touched on above, the level of security of an individual VE will be driven primarily by the organizations using the VE and the context in which the VE will be used. Consideration must be given to tradeoffs such as the cost to develop and maintain the system, system performance as determined through quality of service measurements, and the risk and consequences of a security compromise.

Some VEs will need high levels of security, and run on trusted systems with mandatory access control policies and multiple authentication protocols, while others may not require anything at all and are completely open to all potential users. Most, however, will require some form of minimal-to-medium levels of security controls due to low risk or consequences associated with the misuse of these systems. Examples follow:

Low-level example: An example here might include an unclassified technical trainer for tasks such as vehicle repair. Misuse of this system would not be life threatening or financially onerous; therefore high levels of security might not be cost-effective, and create an unnecessary burden in their use.

Mid-level example: Examples in this category include online game playing. Companies receive significant revenue for providing these services to the public, but they are also popular targets for malicious or over-enthusiastic

hackers. Lack of security can result in degraded game play as some participants exploit security weaknesses in the system. As a result, overall enjoyment of the game by the public is reduced, and the service provider's revenue suffers accordingly.

High-level example: Examples in this category might include a VE intended for use by battlefield commanders for information visualization during a conflict. The information provided may be used to develop sensitive tactics and decisions, and would require protection; but a requirement for mobility and dynamic capabilities may call for a trade-off in reduced security capability.

Very high-level example: A VE used as part of a national strategic system for visualization of intelligence information. Intelligence sources, sensitive relationships, covert operations could all be gleaned from information available in the system, and thus would require the utmost in security protection.

Two other dimensions to VE security are multilevel security and compartmentalization. For example, a military application of a VE could be designed to allow senior commanders to possess sensitive intelligence information while denying it to personnel at lower levels of the chain–of–command, even though both are present in the same area of the virtual environment.

C. OVERVIEW OF VIRTUAL ENVIRONMENTS

A VE is an environment in which physically separated participants have the ability to see, communicate, and 'physically' interact with each other within a computer-generated world. A feeling of a shared space and time are prerequisites.

In a VE, each user controls one or more entities. These entities are represented inside the virtual world by a visual model that all other users can see and interact with. The design of the VE application determines the level of realism and interaction experienced by the participants through such features as its physics-based model foundation, rendering, and rules of interaction.

This section presents a brief overview of VEs, including their desired characteristics, basic architecture, and quality of service concerns. We conclude with a

brief overview of RTEVEs. For a detailed discussion of what VEs are, and their architecture, design and potentials, refer to [Capps97], [Macedonia97], or [Singhal99].

1. Characteristics of an Effective VE

Capps and Stotts [Capps97] list attributes of an effective VE architecture, classifying them into four categories: network topology, interoperability, composability, and rapid evolution. They stated the difficulty of simultaneously addressing attributes in all four categories, and that the then-current examples of VE architectures were deficient in one or more of the categories. This evaluation still holds true today, and may continue to do so.

Network Topology. A good topology will allow for large, if not infinite, scalability in the number of participants in the VE. It will allow for a graceful degradation of the simulation, in the event that any network resource is lost. It must also ensure adequate performance for each participant regardless of the communication capabilities they possess (e.g., T1, ADSL, and modem).

Interoperability. The ability of one VE to transfer an object to another VE without the loss of information is a highly desired attribute. Control of the object must be transferable, including the physical and behavioral properties. If an object explodes in one VE after a certain sequence of events, then the same object should be able to explode in the other VE given the identical circumstances.

Composability. It should be possible to easily create a VE through the union of two separate VEs. The new VE's functionality would be comprised completely of the functionality sets of the two original VEs. This resulting VE would literally be a union of the two parent VEs in every way, both at start-up and during runtime, without undesirable emergent properties.

Rapid Evolution. The ability to rapidly incorporate new technology into a VE with a minimal degree of modification to existing components is essential for ease of modification, research, and expansion. For example, creating a module with the new desired behavior and simply adding the module to the VE with little modification elsewhere in the application.

To date the only attribute that has been reasonably well implemented is that of a network topology.

2. Special Case: Runtime Extensible VEs (RTEVEs)

Traditional VEs are considered non-extensible (i.e., static) in that new types of objects and functionality cannot be incorporated in them while they are executing. In order to add new functionality and objects to conventional VEs the application must be halted, the new item added to the database or application, and then application restarted. In contrast, an RTEVE permits the runtime introduction of new objects and functionality, thus allowing for the runtime extensibility of the system, without stopping and restarting the application. If an executing application has a need for previously unknown capability or object the VE's database can be updated with the new information, and the VE application can load this information at runtime. New code components can be loaded from the database, and therein lies the major new vulnerability of RTEVEs: code modules may maliciously attack the VE. However, this extensibility trait is essential for VE applications that cannot be halted to update their capabilities and data sets. To date, the only visual RTEVEs in existence are hosted within research institutions.

3. Architecture

A distributed VE is comprised of four basic components: copies of the VE application, workstations, database(s), and a network. In general, there are multiple copies of the VE application residing on multiple workstations that tap into a database for information and share data over a network. The data that is to be shared may be a combination of administrative communications, entity-data updates, and streaming video, audio, or other data. For maximum effectiveness and utility, the overall architecture must allow for unrestrained data sharing and operation within the QOS constraints that are decided upon by the developer or user of the application VE. These QOS constraints are discussed in detail in section C.3. of this chapter.

VE Application. A VE application must be able to accurately maintain state information for however many entities are present within the area of view of the host entity of that application. It must correctly maintain each entity's state, respond appropriately to user input, and display accurate views of the portion of

the environment we are interested in. Finally, it must also manage the transmission and reception of data and support communications.

Workstations. Every VE application must run on a workstation, each one equipped with appropriate networked multimedia capabilities (e.g., network connection, graphics card, and sound card).

Database. There must be one or more databases or repositories, either centralized or distributed, that contain data needed for every application should have access to. This information is used to create the environment (e.g., terrain, structures) and every possible object that can exist.

Network. The design of the network communication infrastructure is crucial to the workings of the VE. An infrastructure design is application dependent; that is, the needs of the VE will determine what the design will look like (e.g., reliable vs. unreliable data communications). In general, there will be administrative processes that require reliable communications, typically in a server-client based structure using TCP/IP communications. Likewise, entity state updates that require a constant transmission of entity state protocol packet data units (ESPDUs) between all participants might find a multicast protocol useful, especially in environments where participant numbers is large, and the loss of some packets will not impact the performance of the system because newer packets are not far behind.

4. Quality of Service Concerns

The illusion of real-time interaction is a requirement for a VE, an effect that is sometimes described as *presence.* This illusion is influenced by the rate at which a VE's screen representation is updated and the degree to which interactions with objects in the VE appear to be instantaneous and natural. If the entity update rate is too slow, the VE appears to be jerky and therefore the user lacks a sense of presence. The generally accepted standard update rate for a VE is 30 Hz; at lower rates the human eye begins to notice non-continuous motion. [Grabner01] This originally required that an entity's state information be updated at the same rate as the VE. However, it was soon recognized that the visual representation on the screen could be decoupled from network entity state

updates via dead-reckoning algorithms or other techniques. Since entities tend to keep doing what they're already doing, we can make intelligent guesses about their current state even without constant entity state updates from the network. The use of network-based entity state updates in general means that network QOS has a significant impact on perceptions of presence. The basic network QOS concerns as identified by [Black00] are: bandwidth, delay (also known as latency), jitter, and traffic loss. This thesis concentrates on the QOS areas of Bandwidth and Delay.

a. Bandwidth

Bandwidth is quantity of data delivered by the network to a host per unit time. Different network hardware technologies have different speeds. In common public usage at this time the bandwidth available ranges from 56 KBS dial-up modems to 1 gigabit per second Ethernet connections. The choice of which network technology to use depends on many factors, including cost and distance limitations. VE systems must take these issues into account in order to develop an appropriate data sharing protocol that avoids network bandwidth saturation, and consequently loss of near real-time interactions. The application-controlled items that affect this area the most are the size and number of data packets that are being transmitted within the network.

Generally speaking, more participants in a VE require more bandwidth, since the greater numbers of entities require more state updates. Higher fidelity VEs with more frequent entity state updates can also require higher bandwidth. Some VEs include interactive audio or streaming video, which can tax the bandwidth budget. All these factors must be taken into account when designing the capabilities and architecture of the VE.

b. Delay

In order to maintain an acceptable level of synchronization between the participants of the VE, delay has to be within acceptable limits. Delay can be broken down into two types: network delay, and application delay.

Network Delay. Network delay is the amount of time that a bit takes to pass through the network from one workstation to another. It is the delay induced primarily by the constraint of signals traveling through the network. This varies depending on the network technology (e.g., lasers, fiber optic lines, copper lines,

satellite radio transmissions). Network delay is bounded at the lower end by the speed of light, which is approximately 8.25 milliseconds per time zone [Cheshire96]. Communications via geosynchronous satellite require a round trip to orbit and back, a distance of about 50,000 miles or about 500 milliseconds of latency. This is only the theoretical lower bound; actual network delay is often much higher due to delays introduced by the networking equipment or communications that occur at less than the speed of light. Further details can be found in [Comer00].

Application Delay. Application delay is the amount of time that the workstation itself takes to process the information from the point of identifying the necessity to transmit the data, to the time a packet is formed, is processed by the operating system, and is actually placed on the network. Consequently, it also includes the time between the packet's receipt from the network by the destination host, to when the actual data is received by the application's function that requires the needed data; this period includes the time needed to process the packet, and remove and hand the data to the necessary module in the application. Every step that the data must go through when in the operating system and in the application will induce added delay. Excessive cumulative delay can make it difficult to maintain a synchronized interactive VE.

Latency and delay must be kept within an acceptable, pre-established window that will ensure the desired level of world-consistency and feeling of presence; the illusion of real-time interaction is difficult to maintain if these are too great. As discussed, there are many factors to take into account when determining the possible latency and delay that may be present in a system.

D. OVERVIEW OF AN RTEVE (NPSNET-V)

This section provides a brief history and introduction of NPSNET-V. Here we dissect NPSNET-V into five functional areas that are of concern when dealing with information assurance.

1. Background

The NPSNET program of the Naval Postgraduate School started in 1990 as a research platform for networked virtual environment technology. It is now in its fifth

iteration and known as NPSNET-V. As stated by McGregor and Kapolka, the dream of NPSNET-V is for it to be "…a framework for fully distributed, component based, persistent, networked virtual worlds, extensible at runtime and scalable to infinite size on the Internet." [McGregor01]

In the course of designing and implementing the original architecture of NPSNET-V, the developers realized that a unified hierarchical component framework was required to realize the goals of the program; this prompted a complete change in the structure of the application. For details on the original architecture of NPSNET-V, refer to [Washington01] and [Wathen01].

A full description of the NPSNET-V architecture and interactions would be too extensive for the scope of this work; therefore, only an overview of the component areas that play into the security scope will be covered. For a more detailed description of the program and the current architecture of NPSNET-V, refer to [Capps00], [McGregor01], and [Kapolka02].

2. Functional Component Areas

Programmed using the Java object-oriented language, NPSNET-V is not a virtual world system itself, but a component-based framework used to build virtual worlds by combining functional modules in manners that produce desired characteristics. The run-time extensibility of NPSNET-V is achieved through the ability of incorporating these functional modules into the system during runtime without the need to halt the system, and without prior knowledge of the individual module behaviors. This allows for entirely new behavior to be added to the VE 'on-the-fly.' In terms of exploitable areas, NPSNET-V can be divided into the five functional areas identified in Table 1, and explained below.

Area	Description
Configuration Files	The file that contains the 'blueprints' for the VE
Communications	The communication infrastructure of the VE
Database	The database of all necessary data for the VE
Components	The functional code modules that are used to build the VE and its capabilities
Temporal	The time coordination system

Table 1. Areas of RTEVE Functionality

a. Configuration File

Each virtual world built using the NPSNET-V architecture requires an XML configuration file that delineates the behaviors and structure of the world. This file is a template used by the NPSNET-V application to build the internal hierarchical component structure of the desired VE. The file identifies world-state information, as well as the name of components needed for the structure nodes. The application then downloads and assimilates the needed modules into the component framework. An example configuration file and description is contained in appendix A.

b. Component Framework

NPSNET-V uses a component framework to maintain a hierarchical tree structure of interconnected functional components; this tree is anchored about the 'kernel' of the application, which contains a common, basic set of services primarily related to loading new components. An XML initialization file that is unique to each application loads components necessary for that application. During runtime, the capabilities of the application can be extended by the incorporation of new component modules into the framework as they are needed.

These components include modules such as *models*, which represent the abstract internal state of entities; *controllers* that are used to implement communication protocols;; *views*, which are responsible for visually displaying objects. Since the application is built on Java technology, Java archive (Jar) files are used to hold the class objects for these modules.

During execution, an application may require a previously unknown component. Armed with the name of the component, the application communicates with an LDAP server and retrieves a URL that identifies the location of that component's Jar file. It then retrieves the component and incorporates it into the runtime component structure.

Figure 1 depicts an example NPSNET-V application with the hierarchical component structure. This is a visual depiction of the application created by the configuration file in Appendix A. Note that the 'base.xml', 'gui.xml', and 'dis.xml' are included configuration files, but their tree structures are not depicted.

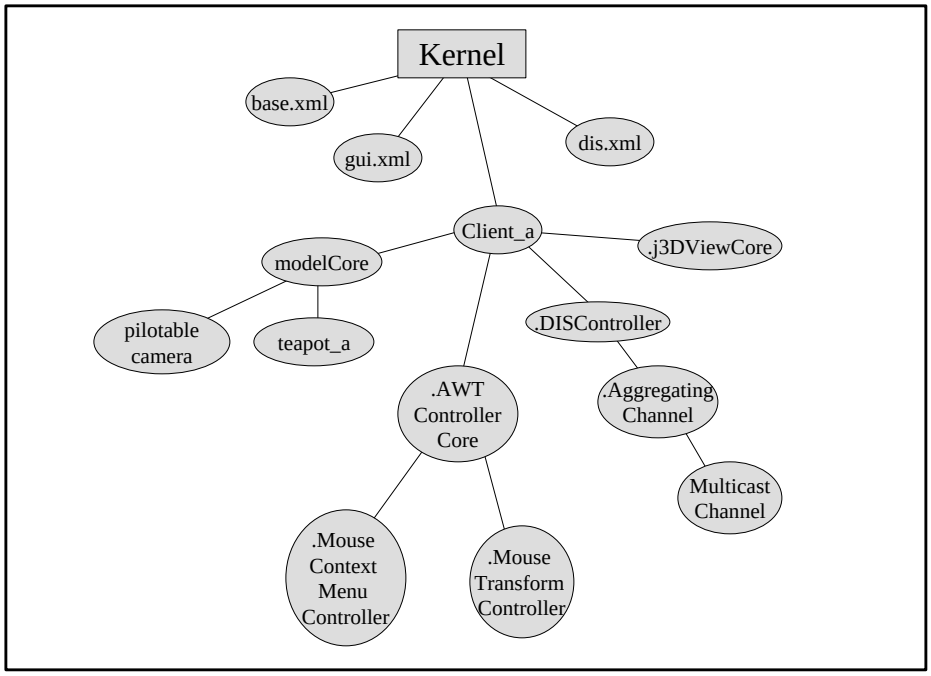

Figure 1. Example NPSNET-V Application Structure

c. Database Architecture

The requirements and capabilities desired in any particular VE will drive the requisite database structure to support it. In general, there are three types of databases required. The first maintains the XML configuration files. The second are LDAP servers that contain the URLs needed for component discovery. The third is the support database structure comprised of multiple HTTP servers that contain the component archive files, and specialized servers providing data such as audio/video streams and terrain data.

d. Network Communications Architecture

Multiple types of communication channels are formed throughout the course of an application's life; from reliable TCP connections to unreliable UDP connections, including multicast and broadcast. These communication paths fall within two groupings. The first revolves around administrative communications, which deal with required exchanges necessary for the proper functioning of the application. These will generally always be reliable TCP type connections and include component/XML file communications, as well as identification and authentication (I&A) traffic.

The second style of communications is that of peer-to-peer entity state update transmissions over reliable or unreliable communications methods such as multicast or UDP connections. These entity state update communications are the heart of the distributed nature of a VE, and often contain information such as the entity position, orientation, or speed. In the Distributed Interactive Simulation communications protocol defined by IEEE standard 1278.1, a packet that contains this information is referred to as an Entity State Protocol Data Unit, or ESPDU

The third form of communications includes the passing of object code modules, terrain data, and streaming audio/video from HTTP and specialized servers. These data will also be transmitted over a combination of reliable and unreliable connection methods.

Figure 2 depicts an example of the network connections that may be formed during a typical session. The dotted lines indicate unreliable multicast channels used to transfer entity state updates and interactions between peers. Solid lines represent the transient reliable connections used to download resources from the World Wide Web, to store and retrieve configuration data from the LDAP service, and to access custom servers.

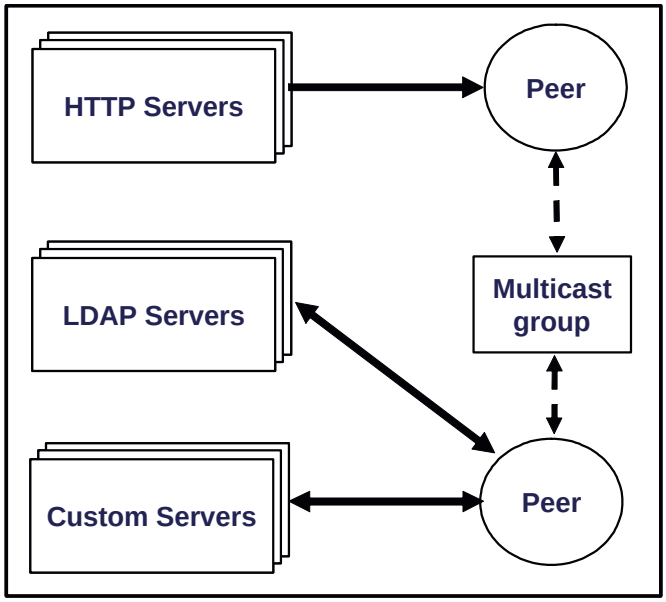

Figure 2. Overview of Network Connection (From Ref [Salles02])

e. Temporal Information

In order to create a synchronized space in which participants can accurately collaborate and interact, a shared 'time-space' must exist. Hosts on TCP/IP networks typically have system clocks, but these clocks are notorious for not being synchronized. The Network Time Protocol (NTP) is a standards-based service that can be used to synchronize system clocks distributed on the Internet to a common time [Mills99]. A *TimeProvider* module can be incorporated into the NPSNET-V framework; this module will supply a time service based on however the module is implemented, typically by querying the system clock on the local host.

E. VE SECURITY EFFORTS AND RELATED TECHNOLOGY

Computer, network, and database security have all been the subject of much research. However, the ever-increasing complexity of computers and the perseverance of hackers ensure that these will continue to be relevant areas of discussion. General network and information assurance issues that transcend systems have been well researched; refer to [Jayaram97] and [Landwehr94].

As for VE security, limited research has been performed. Deployed systems often either ignore security completely or run on trusted networks and hosts that have limited access. As VEs have become more widespread and developed revenue-generating business models security has become more important. In the research community, VE security has generally been treated as an afterthought or of low priority in relation to other issues, such as performance and reliability. SIMNET [Singhal99], which had a genuine security requirement, addressed security through measures external to the VE. Bamboo [Smith00] is an exception; it included digital certificates to authenticate components loaded across the network. The development of viable business models in the environment that the research community ultimately serves, along with advancing hardware and the ubiquity of the unsecured public Internet, have increased interest in VE security. Research and training venues in the past have been protected via physical computer security and identification and authorization schemes inherent in the host operating systems, as with SIMNET. Moreover, some research has been conducted in the area of distributed computing security, which shares many characteristics with visual VEs such as a distributed model in which every host may contain some resource that must be

shared with other participants for the proper functioning of the system. Particularly notable in this field are the efforts surrounding Grid computing, which will be discussed in section E.2.b of this Chapter.

We now turn to an overview of areas of security research that are applicable to VEs. It must be noted that this is not an exhaustive listing, but a representation of the subject domain.

1. Information Assurance Overview

As depicted in Table 2, IA encompasses the five areas: of secrecy, integrity, availability, non-repudiation, and authentication [NSTISSC00], [DOD96]. All five remain concerns throughout the life cycle of a VE. Each one will be addressed to varying degrees through the security policy that is decided upon by the owner of a VE or its user.

Area	Description
Integrity	Prevent unauthorized modification of data
Confidentiality	Prevent unauthorized viewing of data
Availability	Ensure system/data is available for its intended use
Non-repudiation	Ensure a user cannot refute information they placed into the system
Authentication	Ensuring a user/module is who they say they are

Table 2. Areas of Information Assurance (IA).

a. Integrity

This area is concerned with preventing the unauthorized modification of data. Data can be maliciously modified in attempts to destroy the data, mislead users of the data, or, in the case of executable code, perform functions that were not intended by the original author.

b. Confidentiality

This area is concerned with preventing the unauthorized viewing of data. Data in memory or flowing in networks is available to be retrieved/intercepted and viewed. Sensitive information must be protected from this possibility. In some situations this may include information in IP packet headers that contain information on sending and destination IPs. Through traffic analysis, attackers can determine a large amount of

information about a network, such as which nodes are more important than others, which ones are active at any one time, and the level of activity of the organization.

c. Availability

This area is concerned with ensuring the availability of a system and its data for the intended use. If the system is disrupted by an outage, equipment failure, or disruptive denial of service attack then the system is no longer functional and provides no benefit.

d. Non-repudiation

Non-repudiation refers to the inability of an entity to deny having performed some action, or have provided some piece of information. In the case of IA, this refers to being able to legally hold users accountable for information that they provide. It also encompasses the inability of a recipient to deny having received a piece of information.

This area is concerned with preventing a user from providing information, and then later deny having done so. This issue has special importance in systems that use provided information for sensitive operations, especially ones with legal ramifications, and accountability is an issue.

e. Authentication

This area is concerned with ensuring that an entity or component is actually who/what they identify themselves as. This is the prime way of ensuring access only by authorized users. There are three ways to authenticate: through something that is possessed (e.g., smartcard), through something that is known (e.g., password), or through something that the entity is (e.g., biometrics) [Liu01].

When deciding upon a security policy and what mechanisms to implement, consideration of their impact on the QOS concerns previously mentioned must be taken into account. If, for example, encryption is decided upon for confidentiality of network communications, then consideration must be made to the fact that any form of encryption will induce some latency on the transmission of ESPDUs.

2. Security Research Efforts

There has been much security-related research performed, and a comprehensive listing and examination of them is outside the scope of this thesis. Therefore, an effort

has been made is to identify the good portion of the available research that is pertinent to the problem domain of VEs. The following is merely an introduction to these efforts.

a. On-line Game Industry Efforts

An industry that relies greatly on VEs and has to deal with a myriad of security issues specific to their context is that of the on-line game industry. Their issues with security revolve mostly around denial-of-service (DOS) attacks and a plethora of hackers/cheaters. Their problem is an interesting one in that they need to balance the need to be available to anyone interested, but still protect the gaming experience of the honest players. Various mechanisms have been employed by games such as *Ultima*™ and *Age of Empires*™, but many are proprietary and protected from publication. For more information concerning these efforts, refer to [Pritchard00].

There is an area of the on-line-game industry that is akin to visual RTEVEs; that of Multi-User Dungeons (MUDs). These are essentially text-based RTEVEs that have been in existence for over twenty years [Curtis94]. Other than being a form of VE, there similarity with RTEVEs rests in the characteristic of run-time extensibility, as new rooms, objects, characters, and behaviors can be added to the world without turning it off. Some MUDS are provided rudimentary security through the need to register in order to have access to the MUD [Albert94].

b. Grid Computing Technology

[Foster01] defines the Grid problem as "…flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources…[what can be referred to as] 'virtual organizations'". The resource sharing that is referred to goes beyond the traditional idea of data transferring. This concept actually refers to the ability of direct access to data, computing resources, and applications at remote systems/networks for the purpose of solving a problem. This entails interconnecting and resource-brokering amongst many different entities, each of which will want control over just what is shared, with whom it is shared, and to what extent it is shared. This creates numerous complex security concerns, such as authentications, authorization, and access control.

Security efforts in Grid technology are ongoing. This technology tends to incorporate other basic security technology into a comprehensive system for Grid

distributed computing. These efforts have primarily focused on authentication, access control, integrity, and confidentiality. Refer to [Foster98a], [Foster98b], and [Foster01] for more information and details on Grid computing and its associated security efforts.

The distributed nature of the resource sharing technology in Grid computing has potential for being the base of an RTEVE infrastructure. Consequently the associated security mechanism would also have applicability.

c. Symmetric and Asymmetric Encryption

The two standard forms of encryption used to protect data both on a computer and over network communications. Symmetric encryption deals with one key that both parties possess while asymmetric deals with two different but mathematically related keys (public and private) in which each key can be used to decipher data encrypted by the other. Symmetric encryption is much faster than asymmetric, but requires an already secure infrastructure for key distribution. Asymmetric encryption, on the other hand, is much slower than symmetric encryption, but distribution is much easier. Refer to [Simmons79] for more detail.

Public key infrastructure (PKI) is based on asymmetric encryption and is the predominant method of secure communication over the Internet. Refer to [Younglove01] for an overview of PKI.

d. Intrusion Detection Systems (IDS)

These systems are designed to identify malicious activity, preferably as it is occurring. There are two forms of IDSs: signature-based and anomaly-based. Signature-based systems function by monitoring activity on a network, either real-time or through an audit log, and looking for a pattern that matches known intrusion signatures. Anomaly-based systems are designed to identify malicious activity, novel or known, by looking for departures from the known normal operating behavior of the system. Anomaly-based systems hold the greatest promise for complete coverage of intrusion detection, but the complexity of the problem domain is quite high and thus these systems are still not quite robust. Please refer to [Forrest97], [Stillerman99], and [Vigna98] for examples on IDS research.

e. Access Control

The need to constrain entities to only those resources which they are authorized to be exposed to is an important attribute for sensitive applications that involve collaboration between groups of varying security authorizations. Access control mechanisms have been implemented that address this issue directly in the context of CVEs, VOs, and other non-visual VEs. Please refer to [Bullock99] and [Pettifer01] for example implementations.

f. Watermarking

Interesting research in the area of 2-D image and 3-D model watermark technology has been performed. Please refer to [Benedens99] and [Berghel97]. This technology holds some promising possibilities for future data integrity applications within VEs; for instance: the possible development of code module watermarking techniques that could be used to identify authorship or ownership, or even to identify a modified code module.

g. Object Signing

A subset technology of PKI, it can be used to 'sign' objects in order to ensure their authentication after transfer through a network and can be used as a method for ensuring non-repudiation of those objects. [Smith00] details an implementation of this technology for the BAMBOO system.

h. Secure Multicast

The use of multicast in a large-scale VE is almost a requirement. A fair amount of research into secure multicast capabilities has been performed to address the concern of confidentiality. Much of this research concentrates on the infrastructure needed for cryptographic key-distribution to all participants. Please refer to [Molva00] and [Abdalla00].

i. Message Digests and Message Authentication Codes (MACs)

Message digests and MACs are used for data integrity verification purposes. In the case of a data packet, a message digest is produced by running the packet's contents through a hashing algorithm. This algorithm produces a signature (digest) that is usually unique to the given data. The digest is then sent with the data. The receiver of the packet then also runs the packet's data contents through the same

engine, and produces another message digest. If this new digest is different from that provided by the sender, then the data is deemed to have been modified and is not the authentic message. This is usually only effective for non-malicious modifications. An entity that desires to purposely modify a message would just need to generate a new message digest and replace both the digest and the message in the packet, and the message would be accepted.

MACs are similar to message digests; however, the algorithm uses a symmetric cryptographic key for either encryption of a message digest, or for the computation of the digest itself. Both the sender and receiver must possess the same key to develop the matching MACs; otherwise, authentication is impossible. While this ensures message integrity, it does not ensure non-repudiation, since more than one entity has access to the key used to encrypt the message digest. For more detailed information please refer to [Stallings99] and [FIPS198].

The security of message digests can be considerably increased through the use of public-key encryption by encrypting the original message digest with a private key, thereby 'signing' the message digest. A malicious entity would need to have possession of the sender's private key in order to 'sign' a new message digest of the modified data. This adds the feature of non-repudiation in addition to maintaining integrity, and thus is sometimes called a 'digital signature.'

F. TECHNOLOGY USED FOR NPSNET SECURITY MANAGEMENT SYSTEM (NSMS)

1. Java Application Programming Interface (API)

As stated earlier, NPSNET-V is programmed using the JAVA API developed by SUN Microsystems. It is an object-oriented language that, when compiled, produces byte code that can be run on any machine-type that has the Java Virtual Machine installed. This cross-platform capability makes it desirable for large-scale distributed systems such as VEs. For a more detailed discussion of Java refer to the Sun Microsystems Java website at http://java.sun.com, or [Flanagan99].

2. Java Security API

The Java language provides many capabilities for use in protecting Java-based applications, as well as protecting host operating systems from java applications. Since

the purpose of this work is to concentrate on application-to-application interactions, the security of the operating system from java applications is assumed to be adequate and therefore not implemented.

This API, and several extensions to the API, also provides numerous mechanisms for information assurance efforts; including: a Secure Socket Layer (SSL) implementation, Public/private asymmetric and secret symmetric key generation using common key generation algorithms, and message digest and MAC engines. The two extensions that were primarily used in this work are: the Java Secure Socket Extension (JSSE) and the Java Cryptography Extension (JCE).

a. Java Secure Socket Extension (JSSE)

JSSE is an API used in the creation and implementation of Secure Socket Layer (SSL) sockets for encrypted reliable Transport Control Protocol (TCP) server-client communications over the Internet. Using PKI-based asymmetric public/private keys and certificates, the server and clients authenticate each other. During the ensuing handshake, they agree upon a particular symmetric key to use for the remainder of the connection, thus allowing for speedier secure communications.

In order to create SSL connections, each member of the connection must have a public/private key pair, and a certificate that contains the public key that is signed buy a certifying authority. The Java Security API provides a tool called the 'KEYTOOL' that is used to create a 'KeyStore' and the public/private keys, and a 'TrustStore' that holds the certified certificates that are trusted by the owning application. The 'KEYTOOL' service is also used to generate the certificates that contain the public key. For more detailed information on the creation of these items and the associated keys, please refer to [Scott01].

b. Java Cryptography Extension (JCE)

The JCE is the API used for the creation and manipulation of cryptographic keys, both public/private asymmetric keys, and secret or symmetric keys. In particular, this API was used for the generation of symmetric keys used in ciphering features of the NSMS. These keys are generated using one of many different algorithms that is provided to the key generator object and passed to the cipher engines.

The use of the Java Security API and its two Extensions were vital to the implementation described in Chapter 4 of this work. These APIs are robust and allow for inclusion of expanding security packages from outside security providers. For more information refer to the Sun Java web pages at http://java.sun.com/security, and [Scott01].

3. Extensible Markup Language (XML)

XML is not really a language, but a standard for creating languages that meet XML criteria. XML is a hierarchical, extensible meta-data based markup language. It is hierarchical in the sense that the data is structured in a parent-child, tree-style fashion. It is extensible in the sense that the developer of an XML-based language can increase the type of different data that his language can function with. In addition, it is meta-data based in the sense that every piece of data in the file must be 'described' by using a descriptive tag that is associated with the particular data item. For more information on the details of XML, refer to [Hunter02].

As stated in earlier sections, NPSNET-V is a framework architecture on which an RTEVE is built. XML-based configuration files provide the hierarchical blueprint needed for the proper initialization of a desired NPSNET-V world.

G. SUMMARY

This chapter provided an overview of VEs and RTEVEs, discussing their architecture and the QOS concerns that affect them; this discussion concluded with a breakdown of NPSNET-V into its five functional areas. Security of VE was also introduced, along with a discussion of why it is an important area of research. The five areas of Information assurance were presented, as well as brief discussion on several security-related research topics that are applicable to the security of VEs. This section concluded with a brief overview of the technology that is pertinent to the development of the NSMS that will be presented in Chapter IV.

The next chapter presents a taxonomy of RTEVE security areas, based on the research covered in this chapter. It is followed by two matrices that can be used to better understand weaknesses in the identified areas, and possible mechanisms that can be used to address them.

THIS PAGE INTENTIONALLY LEFT BLANK

III. TAXONOMY OF RTEVE SECURITY

This chapter provides a thorough review of security issues within the realm of RTEVEs, using the NPSNET-V RTEVE discussed in the previous chapter as a case study. It first presents security concerns with relation to the five areas of RTEVE functionality, and then to the five areas of IA. Subsequently, it will present a taxonomy of RTEVE security areas through the form of a 5x5 matrix. Two subsequent matrices will present scenarios that will clarify the areas, and mechanisms that can be used to address the identified areas. It should be noted that these matrices are merely the beginning of a taxonomy for RTEVEs. Further research may eventually reveal that this taxonomy should either be expanded or contracted.

A. SECURITY DISCUSSION OF RTEVES

This section will present two discussions on security concerns with respect to the functional component areas of NPSNET-V and the five areas of IA. These discussions are not meant to be all inclusion and comprehensive, but merely a sample representation of the concerns in the indicated areas.

1. Security Relative to the Five Functional Component Areas

This section reiterates the five functional areas of RTEVEs, as based on the architecture of NPSNET-V, and introduces a brief introduction of security issues relevant to each one.

a. Configuration Files

The security of configuration files deals primarily with their modification/replacement or deletion. Deletion of the file would prevent the ability to use the VE for its intended purpose. Modification of configuration file could be performed to include a malicious module that is then loaded by the applications and executed. Replacement of the file with another file affects the availability of the intended VE, or may introduce unwanted behavior into the system.

b. Communications

Security of communications revolves around the protection of the communication paths and the data that travels on these paths. This includes the confidentiality, authentication, and integrity of the data in the network, as well as the

information that the traffic of the network can convey through analysis efforts. Non-repudiation plays a factor when accountability of what is communicated is of concern.

c. Database

Security of the database deals with the protection of the data stored within the database architecture of the system. This includes all portions of the VE that require storage (e.g., Jar files, terrain data, streaming audio/video data).

d. Components

This section deals specifically with the security of the application kernel and the individual byte-code component modules that are used to build the VE applications. Since these are executable modules, inclusion of malicious functionality would have far reaching consequences, especially if the size of the VE is extensive and all applications are downloading it; a virus could be spread almost instantaneously to potentially thousands of users. Also, if a needed module has been deleted, the availability of the system would be degraded because the desired functionality is no longer present.

e. Temporal

A common, coordinated time-space is critical if the real-time synchronization of participant interaction is desired. By manipulating the individual 'time-space' of some participants and not others, a malicious entity can effectively destroy the temporal consistency of the VE, thus reducing or destroying its interactive capability.

2. Security Relative to the Five Information Assurance Areas

This section reiterates the five areas of IA, and provides a discussion on their relevance within RTEVEs.

a. Integrity

Integrity of the areas of an RTEVE is of primary concern. If components and data of a system can be modified, then confidentiality and availability can easily be subverted by a knowledgeable hacker, either through direct manipulation, or the use of a virus/worm.

Integrity of the XML configuration files is crucial for the proper operation of a VE. Any unauthorized modification of its contents would alter the intended functionality of the VE, and possibly cause malicious code to be incorporated into the VE.

The extensible nature of RTEVEs makes them vulnerable to maliciously modified modules. If a trojan horse were substituted for a legitimate component during transmission or in the database, all receiving applications would be subverted, resulting in undesired modified behavior.

The integrity of ESPDUs and temporal information are also a concern. By modifying the state information of participating entities, a malicious entity can 'drive' the information displayed on systems and provide misleading information. The modification of temporal data can be used to reorder events, reducing the effectiveness and usability of the system.

Another type of integrity attack is the so-called 'replay attack', in which the attacker records legitimate traffic, then at some later time resends exactly the same packets. Since the packets were at one time legitimate and were encrypted with a correct encryption key, the receiving system may mistakenly accept them, resulting in the ability of the attacker to repeat past behaviors in the VE.

b. Confidentiality

The possibility of RTEVEs being used in sensitive applications necessitates that confidentiality be a concern. Within this context, secrecy of both computer and network systems must be taken into account. This area has an impact in the integrity aspect as well. For instance, if an adversary is able to acquire a copy of a module/file, they then can intelligently modify the component to include behavior that may be detrimental to the system.

In the case of military applications, there will be times when participants of varying classification levels will have need to coexist in the same virtual world, and thus have access to different information based on their clearance level. Access control mechanisms come in to play in these situations.

Information can be gleaned by the analysis of traffic flowing across the network. Depending on the purpose of the application and current state of affairs, an increase in data transmission can signal preparations for a military action or reveal relationships between units. Even if the data is not readable, an attacker can determine what participants are most active or most informed and use that information to determine target vulnerabilities.

c. Availability

The availability of an RTEVE could be of great importance to an organization that is using the system for time-sensitive operations, such as battlefield information visualization. The loss of use of a system such as could result in poor and deadly decision. Loss of availability could be as a result of a distributed Denial of Service (DOS) attacks, such as packet saturation of the communication paths; introduction of code designed to shut down one or more applications; or even attacks directed toward physical components of the RTEVE, such as data links or workstations.

d. Non-repudiation

In RTEVEs that are used for sensitive, critical information, the source of information that is injected into the system must be able to bear responsibility for that information. The ability to assign responsibility to a source can be used to identify possible malfunctioning equipment; but, more importantly, in can be used to identify deliberate misinformation.

In the context of a battlefield visualization system, if a commander gives the order to launch missiles on what the system shows is a hostile aircraft, but in reality is a passenger jetliner, the source of the information must be traceable and must not be able to repudiate its introduction of the misinformation. By being able to track down the cause of misinformation, vulnerabilities to the system can be identified, and those responsible can be held accountable.

e. Authentication

Methods such as IP hijacking or spoofing could be employed by a hacker to enter into the RTEVE and observe and interact with participants. The possibility also exists that a malicious entity acquires a copy of the VE applications and attempts to join a VE in the hopes of passively viewing the contents of the VE. Authentication of

component modules is also a significant issue; we may require authentication of components to assure that an attacker has not substituted a modified component while the component was transiting the network. The nature of RTEVEs would require the need to authenticate users across a distributed computing environment; necessitating, for example, the incorporation of PKI into the architecture.

B. RTEVE SECURITY TAXONOMY

This section is divided into three sub-sections. The first introduces a taxonomy of twenty-five RTEVE security areas developed through an analysis of NPSNET-V and IA security areas. The second subsection contains example security attacks and scenarios used to give the reader insight into what types of issues can be associated with each one of the individual areas. The final sub-section identifies various researched security mechanisms that can be applied to each of the twenty-five areas.

1. RTEVE Security Areas

This section introduces a taxonomy that was developed by combining the five functional component areas of the NPSNET-V RTEVE and the five IA areas. Each of the five functional areas was considered as a target by each of the IA areas. This resulted in twenty-five different security areas that are presented in matrix form in Table 3, below. Each area has a corresponding identifying code that is individually explained following the matrix.

	Integrity	Availability	Confidentiality	Non-repudiation	Authentication
Configuration files	CFI	CFAV	CFC	CFN	CFAT
Components	CMI	CMAV	CMC	CMN	CMAT
Database	DBI	DBAV	DBC	DBN	DBAT
Communications	CI	CAV	CC	CN	CAT
Temporal	TI	TAV	TC	TN	TAT

Table 3. RTEVE Security Areas Matrix

a. Configuration Files

The Configuration Files grouping focuses on the configuration files that are used to delineate the basic structure and main components of a desired VE.

- ***CFI***: **Configuration File Integrity**. This area is concerned with protecting the configuration files from unauthorized modification; and, if they are modified, identifying that fact. An attacker could modify the configuration file of a VE to have the applications load a 'malicious' module, or to cause the VE to function improperly.

- ***CFAV***: **Configuration File Availability**. This area is concerned with ensuring that the configuration file is available for its intended use. A configuration file is required for VE initialization; if the file is deleted or modified to prevent its use, then the user can't use the system for the purpose for which it was intended.

- ***CFC***: **Configuration File Confidentiality**. This area is concerned with protecting the configuration files from unauthorized viewing. If a malicious entity is able to acquire a copy of the file, they then have the ability to intelligently create a malicious replacement file. The contents of the configuration file might also give an attacker insight into what is being modeled; the existence of a certain type of aircraft entity in the configuration file could give the attacker an edge in predicting the nature of the simulation.

- ***CFN***: **Configuration File Non-repudiation**. This area is concerned with ensuring that an authorized user that places a configuration file in the system has no way of denying that he did so. If an authorized user intentionally changes a configuration file for some malicious purpose, there must be a capability that does not allow them the ability to deny having made the modification.

- ***CFAT***: **Configuration File Authentication**. This area is concerned with the process of authenticating the identity of a configuration file. There must be a way to ensure that a configuration file in the system is indeed the correct file, and not a modified copy of the file.

b. Components

The Components group focuses on the components that make up an individual RTEVE system; the main application kernel that each user must possess and the plethora of individual modules that are designed for each entity, behavior, and protocol.

- ***CMI***: **Component Integrity**. This area is concerned with protecting the application kernel and modules from unauthorized modification; and, if they are modified, identifying that fact. An attacker could modify an individual module in the system to contain a virus. The virus is then easily spread throughout the system as applications assimilate the module.
- ***CMAV***: **Component Availability**. This area is concerned with ensuring that the application and modules are available for their intended use. If the kernel or component is deleted, or otherwise modified to prevent its proper use, then the application is not able to function as intended and availability of the system is diminished.
- ***CMC***: **Component Confidentiality**. This area is concerned with protecting the application kernel and modules from unauthorized viewing. If a malicious entity is able to acquire a copy of the application or a module, they then have the ability to intelligently create a malicious replacement component. The contents of the component may also require secrecy. The capabilities of a weapon might be deduced from the code that is intended to model its behavior.
- ***CMN***: **Component Non-repudiation**. This area is concerned with ensuring that an authorized user that places a module in the system has no way of denying that he did so. If an authorized user intentionally places a module into the system that contains malicious code, there must be mechanisms to ensure they cannot

refute that action. Conversely if a malicious entity modifies an authorized module to contain malicious code, the same mechanisms should be able to prove the original authorized user was not responsible.

- ***CMAT***: **Component Authentication**. This area is concerned with the process of authenticating the identity of a component, ensuring that it is the correct component. Good CMAT practices will assist against attacks that fall in the CMI and CMAV areas.

c. Database

The database grouping focuses on the database structure of a given VE. A database could be central or distributed; it could contain modules of code or binary data for terrain, or anything that could be needed within a VE.

- ***DBI***: **Database Integrity**. This area is concerned with protecting the information within the database from unauthorized modification; and, if anything is modified, identifying that fact. If an attacker were able to access and modify the database, they potentially can affect every application that uses its data. Equally as damaging would be a virus was able to that access the database and writes itself into every component within the database; when a VE is initialized and the components are transmitted to all users, the virus would massively propagated.

- ***DBAV***: **Database Availability**. This area is concerned with ensuring that the information in the database is available for its intended use. RTEVEs require the ability to access configuration files, unknown modules, and other data for proper functioning. If these are not available for use, then the applications are useless.

- ***DBC***: **Database Confidentiality**. This area is concerned with protecting the information in the database from unauthorized viewing. Some databases may contain classified information, which requires protection. Also, if modules within the database

are acquired, they can be used to design malicious modules that can pass for the modules within the database.

- ***DBN***: **Database Non-repudiation**. This area is concerned with ensuring that an authorized user that places data in the database cannot deny that action. In order to hold someone accountable for any harm that is caused by something placed in the database, there must be accountability mechanisms.

- ***DBAT***: **Database Authentication**. This area is concerned with the process of authenticating the identity of a database. If a malicious entity were to create a database that mimicked the real database, and contained accurate modules with malicious code, or no code at all, then the VE would be subject to attacks in the DBAV or DBI areas.

d. Communications

The Communications grouping focuses on the communications paths of the system and the data that rides on those paths.

- ***CI***: **Communications Integrity**. This area is concerned with protecting the data passing on the communications paths from unauthorized modification; and, if they are modified, identifying that fact. An attacker could modify the data packets of a particular entity to misrepresent that entity's state in other users' applications, making the entity appear in a different position or perform different tasks.

- ***CAV***: **Communications Availability**. This area is concerned with ensuring that the data on the communications paths is available for its intended use, and the communications paths themselves are fully functional and available for their intended use. If a an attacker is able to disrupt a communications link, or flood the communications with useless packets, then the application will be unable to be used for its intended purpose.

- ***CC***: **Communications Confidentiality**. This area is concerned with protecting the data on the communications paths from unauthorized viewing. A party that is interested in gathering intelligence on what is occurring within the VE, but not disrupting it, may attempt to intercept the data as it is traveling on the communications paths and reconstruct what is occurring.

- ***CN***: **Communications Non-repudiation**. This area is concerned with ensuring that an authorized user that places data into the communications channels has cannot deny that he did so. If an authorized user places false information into a VE system, which results in adverse consequences, there is no way to hold them accountable unless a non-repudiation mechanism is in place.

- ***CAT***: **Communications Authentication**. This area is concerned with the process of authenticating the identity of the data on the communications path. The ability of ensuring that every piece of data received off the communications paths is from an authorized user is an important attribute for addressing the CCI and CCAV areas.

e. Temporal

The temporal group focuses on the time synchronization system required to ensure that all participants in the VE are functioning with coordinated clocks. This function is generally provided to the Internet through the NTP (Network Time Protocol) system; individual network/system can routinely update their internal clock to match true atomic clock time received through this system. A dedicated time-server could also be used to provide a common time-space for synchronized interactions.

- ***TI***: **Temporal Integrity**. This area is concerned with protecting the temporal data that synchronizes all entities with a VE. If an attacker individually modifies the temporal information sent to each participant, then they would be functioning in different time

spaces within the same VE, destroying the usability for coordinated interaction.

- ***TAV***: **Temporal Availability**. This area is concerned with ensuring the availability of VE-wide time synchronization data. If the synchronization data from the time servers is blocked from reaching the VE users, then each user's machine error will eventually cause disparity amongst all users as their workstation's clocks begin to drift at varying rates.

- ***TC***: **Temporal Confidentiality**. This area is concerned with protecting the temporal data from unauthorized viewing. If a malicious entity were able to intercept the temporal synchronization to the user applications, they could develop more intelligent temporal attacks such as intercepting and modifying the synchronization data to specific users in an attempt to disrupt only their data or system in a manner that makes temporal sense, but that is not accurate.

- ***TN***: **Temporal Non-repudiation**. This area is concerned with ensuring that the temporal data can be accurately traced back to its originating time-server. An authorized time-server should not be able to refute the synchronization data that it has provided. Also, if synchronization data was altered by a rogue time-server, the same mechanisms should be able to exonerate the authorized time-servers and indicate such.

- ***TAT***: **Temporal Authentication**. This area is concerned with the process of authenticating the temporal data from the time-server. There should be a way to ensure that the synchronization data is from an authorized time-server. Good TAT practices will protect against attacks that fall in the TI and TAV areas.

2. Security Scenarios

This section presents a matrix, Table 4, that identifies possible attacks and scenarios that would provide some understanding to the types of concerns that fall within each of the twenty-five RTEVE security areas. For each attack and scenario, the security areas that are, or can be, affected contain the identifying number (Arabic or Roman, e.g., 3,i) of the corresponding example. Please note that this is not meant to be an all-encompassing list of attacks, as the range of attacks is limitless.

	Integrity	Availability	Confidentiality	Non-repudiation	Authentication
Configuration files	1,9,ii	7,9,10,ii	11,14,ii	4	1,14,ii
Components	2,8,iii	7,8,10,iv	11,14,iii	4	2,19,14,ii,iii
Database	2,7,iii	7,10,iv	12,14,v,viii	4	1,2,14,ii,iii
Communications	3,4,vi,x	6,ix	13,15,i,vi	4	3,4,14,17,18,vi
Temporal	5,i,ix	5,vii	x	4	5,ix

Table 4. RTEVE Security Scenario Matrix

a. Simple Attacks

This category contains simple known attacks and examples that are used to identify the RTEVE security areas that are affected by the attack. Each attack is identified by a number, and that number is placed in the matrix presented in Table 4.

[1] A malicious entity modifies the XML configuration file in the database to download a malicious module.

[2] An entity modifies an application or module located in the database to include malicious code.

[3] An entity that has performed an IP hijack or has subverted a router modifies ESPDUs in order to mislead the recipients of the packets, or replaces modules enroute with malicious code such as a virus.

[4] An entity uses the IP of an authorized user to inject fake packets that create confusion in the VE.

[5] A malicious entity manipulates the timing information that is used for synchronization in order to disrupt the VE and have events occur out of sequence.

[6] A traditional DOS attack is performed by flooding the target computers/networks with useless packets, thereby slowing down or halting the simulation.

[7] The system database is destroyed, thereby removing all ability to locate needed modules/data.

[8] Modules are modified so that their behavior is incorrect

[9] Configuration files are corrupted thereby denying the ability to correctly configure a desired VE

[10] A worm/virus modifies/destroys needed files/modules/data.

[11] A malicious entity surreptitiously acquires a copy of the application or a module to develop malicious copies for future use.

[12] A malicious entity retrieves the password file for the VE system. Then uses a cracking program to break the hashes.

[13] An entity uses a packet-sniffer to acquire unencrypted network traffic containing ESPDUs and administrative messages of the system; allowing them to see what the system is being used for.

[14] A rogue, authorized user attains system information that is beyond their clearance level.

[15] A malicious entity uses a packet sniffer to perform traffic analysis on VE network traffic in an attempt to identify what participants are most active and target them for further action, or to determine the general level of simulation activity, which could signal that some operation may be occurring soon.

[16] An authorized user knowingly places false information into the VE, or modifies a configuration files, application, or component for malicious reasons.

[17] Man in the Middle. A malicious entity inserts itself between the system and a user, in a manner where the user thinks the entity is the system, and the system thinks he is the user.

[18] IP spoofing. A malicious entity uses the IP address of an authorized user to send packets into the system so that it appears the information is from that known user.

[19] A malicious user possesses a copy of the core application and is able to get accepted as an authorized participant

b. Scenarios

This category contains simple attack scenarios that may be performed on an RTEVE system. The areas of attack that are covered by these scenarios are identified in the matrix by the scenario's Roman numeral.

[i] A malicious entity establishes a time-server and manages to have it replace the real time server for a particular VE. He then analyzes the generated traffic on the communications channels and identifies which 'users' are producing the most updates. He then manipulates the time information being sent to those identified systems, causing their synchronization to change relative to the entire VE; this causes the entities of the VE to no longer share the same time-space. Users may or may not be able to identify the miss-synchronization, and poor decision can arise. Additionally, the mismatch may cause a host to reject time-stamped packets for being too old.

[ii] An entity acquires a copy of a configuration file, and modifies it so that an applications requires a module containing malicious code that resides on a database created by the attacker. He then manages to replace the original configuration file with the modified, malicious copy. The configuration file is retrieved by other users and the malicious module is downloaded. The module

turns out to be a trojan horse that contains code that, at a specific time, will cause the system to shut down, thus terminating the VE.

[iii] Having detailed knowledge of specific module used by a VE, a malicious entity develops a virus that will search for that particular module in a database and modify it to contain code that will retransmit every ESPDU received by the host application to a system set-up by the attacker. The attacker then finds some way for the virus to be placed on the system (i.e., e-mail, an unwitting authorized user installs free software, etc.). The virus is executed and the module is modified directly within the database. A VE is then initiated that requires that module, and now every instantiation of the module is transmitting ESPDUs back to the malicious entity. The entity now sits back and watches what is happening.

[iv] A system that contains the database of modules for a specific VE, is infected with a worm that destroys all data on the system. The entire database is destroyed, and there are no other copies.

[v] A virus somehow is placed on a system that contains a classified terrain data server for VE systems. When the virus is executed, it is able to read the data from the server and transmits it back to the virus' creator.

[vi] A malicious entity subverts a number of key routers in the network and monitors the traffic, looking for a particular entity's ESPDUs. When he sees one, he modifies the data in the ESPDU in an effort to manipulate the VE to his own ends, and transmits the modified packet. All receivers of that packet now have an incorrect state for that entity.

[vii] An entity removes the time-server for a system, causing the systems to rely on their own internal clocks. Over time, the time difference between machines grows to a point where coordination is difficult, if not impossible within the VE, and the VE loses its ability to be used.

[viii] In a research and development world, a corporate cyber-spy infiltrates the database of a weapons manufacturer that uses a VE to develop the

weapon in a collaborative environment. He hunts down the information that describes the behavior and performance of the system, then quietly disappears.

[ix] A VE is being used as a battlefield visualization system during a military campaign. The adversary in the campaign identifies the communication paths of the system and manages to disable the necessary routers, thereby denying the use of the system to the field commander.

[x] An American commander uses a battle-field visualization system as an early warning system for incoming missiles. The enemy launches a missile and desires to mislead the US forces in order to enhance the chances of success. The enemy manages to subvert the workstation of the system that is tracking the missile and placing ESPDUs into the VE system. The enemy also can see the synchronization stamp provided by the time-servers. By manipulating the time stamp of the outgoing packets that contain the information on their missile, they cause the commander to believe that the missile is slower than it really is. The commander is then surprised/confused when the missile enters the close-in detection system possibly leading to confusion/uncertainty. Even if the missile is successfully defended against, the commander subsequently loses confidence in the information provided by the system.

3. Security Measures

This section presents a matrix, Table 5, which identifies a sample of available security mechanisms and technology that can be applied to each of the areas. This listing is not comprehensive, but merely illustrative.

	Integrity	Availability	Confidentiality	Non-repudiation	Authentication
Configuration files	c,d,e	f	a,c,d	c	c
Components	c,d,e	f	a,c,d	c	c
Database	c,d,e	f	a,c,d	c	c
Communications	c,d,e	b,f	a,d	c	c
Temporal	c,e	f	a	c	c

Table 5. RTEVE Security Measures Matrix

a. Encryption

Encryption techniques fall within the two different key structures explained in Chapter II, section E.2.c. Please refer to that section for details on symmetric & asymmetric encryption.

Networks can be encrypted in various methods. Two of the most common are end-to-end and link encryption. In end-to-end, the encryption and decryption of the data occurs at the individual workstations themselves, and therefore the packet headers on the network are still subject to observation and traffic analysis can be performed. Link encryption, on the other hand, occurs at the nodes of the network, where each node-node connection is encrypted. In link encryption, all data, including the packet headers, is encrypted, making traffic analysis much more difficult.

b. Intrusion Detection

This covers the ability to detect malicious behavior within the network and workstations. Refer to Chapter II, section E.2.d for details.

c. Identification and Authentication (I&A)

Identification and authentication is the process of authenticating a user or object. The authentication process is normally accomplished with something the user/object 'is', 'knows', or 'possesses'. An example of something a user/object 'is' would be biometric information, such as fingerprints, or hashing signatures. An example of something a user/object 'knows' would be a password. An example of something a user/object 'possesses' would be a token of some sort, such as a 'smartcard' or badge.

d. Access Control Methods

Access control deals with ensuring users are confined to those areas for which they have access. Refer to Chapter II, section E.2.e for details.

e. Modification Detection

Modification detection uses Hashing algorithms to develop signatures for data, which can then be used to identify modifications. Message digests and MACs are examples of this mechanism. Refer to Chapter II, section E.2.h for details.

f. Availability Assurance

This area encompasses the methods of protecting the availability of resources from issues such as single point of failure vulnerabilities. Methods include replication of resources, so that if one copy becomes unusable, other copies are still available; distributing database resources are prime examples.

Important, sensitive systems must also guard for physical availability. Ensuring redundant power supplies are available in the event of primary power loss can ensure that the protected system will always be ready for its purpose.

C. SUMMARY

This chapter presented a matrix of twenty-five RTEVE security areas. These areas were created by pairing up the 5 information assurance areas with each of the five functional areas of an RTEVE as represented by the NPSNET-V framework. For clarity, a second matrix was developed that associates various security scenarios with the respective security areas in the matrix. And, finally, a third matrix was developed to associate developed mechanisms with the security areas that they can be used to address.

The next chapter will provide the details on the design and implementation of the security capability that was developed to address several security areas of NPSNET-V.

IV. NPSNET-V SECURITY MANAGEMENT SYSTEM (NSMS)

This chapter contains the design and implementation details of the basis of a security management system for NPSNET-V. Our design and implementation is not complete, but it was sufficient for prototyping three NSMS security-enabling filters.

A. REQUIREMENTS OF AN RTEVE SECURITY MANAGEMENT SYSTEM

Ideally, a security management system will provide complete coverage of the twenty-five areas of RTEVE security as identified in chapter III. Its predominant characteristics should include the following:

- Must be distributed to prevent single-point-of-failure weakness and also for the ability to scale limitlessly. It should be comprised of a combination of host-based, application-based, and middleware, networked-based security functionality.
- Must maintain a robust and efficient key distribution system for all participants. This system must be able to perform and manage routine key distribution to all participants, to include when a new communication channel is opened that needs to use a key, as well as timely coordinated routine key changes. It must also be able to handle emergency key distribution for when a compromise is detected.
- Ability to identify malicious behavior and provide a response to it. It must possess a robust intrusion detection and response capability; preferably an anomaly-based detection system for identification of unknown and novel attacks and intrusions. The response capability must be able to isolate the intruder from the system, permanently, or even allow for the possibility of decoy mechanisms that can be used to fool intruders while information about them is being gathered for potential use [Michael02].
- Must allow for I&A, integrity checks, and a non-repudiation capability of every possible entity or object that can access resources, resources that can be accessed, or objects that are passed. This includes users, configuration files, applications, modules, and data packets.

- Should allow for the simultaneous presence of participants with differing security levels and compartment authorizations through the use of access control mechanisms.

Some of these capabilities may be sacrificed if there is no need for them in a specific application. As with any software project, features are driven by requirements.

B. SCOPE OF NSMS

This section will cover the scope of the NSMS including the assumptions made and the capabilities that were targeted. But, first will be a quick overview of the RTEVE security areas that are addressed to some extent by this design.

1. RTEVE Security Areas Addressed

	Integrity	Availability	Confidentiality	Non-repudiation	Authentication
Configuration files	CFI	CFAV	CFC	CFN	CFAT
Components	CMI	CMAV	CMC	CMN	CMAT
Database	DBI	DBAV	DBC	DBN	DBAT
Communications	CI	CAV	CC	CN	CAT
Temporal	TI	TAV	TC	TN	TAT

Table 6. RTEVE Security Areas Addressed by NSMS

This implementation of the NSMS addresses, to some degree, each of the identified areas in Table 6. Note that this implementation focuses on ESPDU transmissions, which form only one of several communications between applications and servers. CMAT is addressed through the use of certificates and a PKI to authenticate one type of component, the *StandardSecurityManager*, while leaving unaddressed other types of components. CI is addressed through the use of a message digest provided with each packet for integrity verification. CC is addressed through encrypting data packets. CAT is addressed through the indirect fact that only authenticated applications possess correct keys for encryptions, and if an ESPDU was encrypted with the correct key, then the application that sent it was authenticated, and therefore the ESPDU is authentic.

2. Assumptions

The main assumptions for the areas covered by the NSMS are as follows:

- An adequate computer IA policy is in place, including I&A and access control mechanisms for and within computer and network resources.
- Authorized users of a workstation are also authorized users of the NPSNET-V framework and any worlds created with it.
- Certificates used for SSL authentication are unable to be maliciously acquired.
- Existence of a synchronized, uncompromised time space.

3. Capabilities

The NSMS addresses the four areas identified in Table 6 through a number of mechanisms performed through two separate, yet communicating software entities: A security-focused server and a security-based component system embedded within NPSNET-V applications. The component-based system is further broken down into two types of objects: a *StandardSecurityManager* object and Filter Objects, of which there are three. The mechanisms in these objects focus on packet communications of the NPSNET-V framework.

The server's capabilities are:

- *StandardSecurityManager* component authentication through the use of certificates, and managing SSL connections.
- Generation of symmetric keys for use in packet encryption functions.
- Ability to generate keys using three different key generation algorithms: Data Encryption Standard (DES), Multiple DES (DESede), and Blowfish. [Stallings98] and [Oaks01]. Also, the use of varying key lengths (56 – 448 bits) when using the Blowfish algorithm.

- Management of all applications and associated registered encryption-capable filters and their keys that successfully authenticate and connect to the server.
- Capability of setting an 'Active period' in which the key is to be used, which will allow for coordinated key changes based on time.

The *StandardSecurityManager* capabilities include:

- Management of filter Objects
- Management of symmetric keys that are received from the security server. To include proper routing of a key to the intended encryption mechanism, the proper management of keys provided for future use, and the effective change of keys when a new key's active period begins.

The various filters' capabilities include the following:

- Symmetric key management, and enciphering and deciphering operations on packets.
- Management of sequencing numbers within outbound and inbound data packets.
- Management of data integrity verification operations on data packets through the use of message digests.

C. DESIGN OVERVIEW

1. Technology Used

As discussed in Chapter II, section F, the technology used in the research reported here utilizes the Java API and the Java Security API. The majority of the security functionality was created using the two extensions to the Java Security API: the JSSE for SSL capabilities, and the JCE for cryptographic capabilities. And finally, XML was used in the process of developing several configuration files for use in the testing and experimentation of the NSMS.

2. Patterns Used

"Software patterns are reusable solutions to recurring problems that occur during software development." [Grand98] NPSNET-V makes use of various well-known

programming patterns. The following three patterns were widely used within the design of the NSMS. Refer to [Grand98] for greater details.

a. Filter

"The filter pattern allows objects that perform different transformations and computations on streams of data and that have compatible interfaces to dynamically connect in order to perform arbitrary operations on streams of data." [Grand98] This pattern allows a programmer to develop a number of different objects that manipulate a stream of data in different ways. The programmer can then connect these filter objects, varying the sequence that they are connected, in order to produce the desired sequence of manipulations on the initial data.

This pattern was used in developing the filters that perform operations on the ESPDUs. Further detail will be provided later in this chapter.

b. Interface

The idea behind interfaces is to "…Keep a class that uses data and services provided by instances of other classes independent of those classes by having it access those instances through an interface." [Grand98] This pattern allows for a plug-and-play type of architecture within an application. For example, a certain application relies on a specific service to be provided, and that service can be provided by different service providers. All that the application interface needs to specify is a set of functions that any ServiceProvider must implement in order to provide the desired service. Any ServiceProvider that is to be used with the application must implement the functions required by the interface in order for the applications to be able to use its services.

Interfaces are used widely throughout NPSNET-V. They are the primary means used by most components to couple with each other. An interface permits an object to communicate with another object, without knowing how the other object provides the desired service or the implementation class. As long as the requesting object passes the arguments expected by the interface of the other object, the request can be acted on by the service.

c. Listener

More commonly known as the 'Observer', this pattern lays the foundation for efficient notification of events occurring in one object to be transmitted to interested

objects that register themselves with that first object. In the NSMS this pattern is also widely used in communications between the objects contained in the NPSNET-V framework.

3. Three Main Components

The NSMS is comprised of three major components: A *SecureServer* system, a *SecurityManager* type object and filter objects. The relationship of these objects is depicted in Figure 3 below. The *SecurityManager* is an interface that supplies required methods for objects built on that interface. For this work, the *StandardSecurityManager* object, that implements the *SecurityManager* interface. All extended NPSNET classes and implemented interfaces that are referenced in this section are described in detail in section C.4 of this chapter.

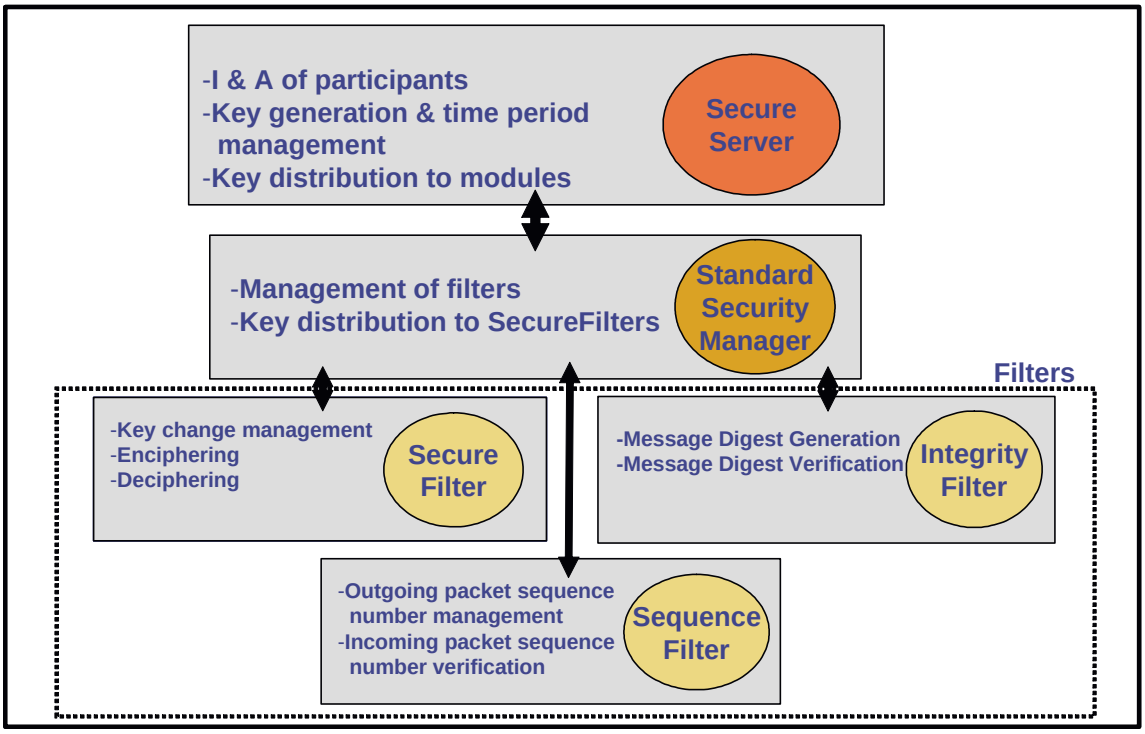

Figure 3. NSMS Main Component Objects and Their Capabilities

a. SecureServer

The *SecureServer* is an object that extends *Module*, allowing it to be a module within an NPSNET-V application. It also implements two Interfaces: *Runnable*

and *Startable*. Runnable is used to spawn off the server's listening process as a separate thread, and Startable is used by the NPSNET framework to start and stop the thread.

Several Objects were designed for use with the *SecureServer* in providing its functionality. It Contains a *KeyMaker* object, which is used for the production of *SecretKeyPack*s. It also contains any number of *SecureServerConnection* objects, each of which represents a communications link to an individual *StandardSecurityManager* object. Descriptions of these objects can be found in section C.4 of this chapter.

The server's functionality can be broken down into three areas; these are: management of connected *StandardSecurityManager*s, management of symmetric key generation, and distribution to registered *SecureFilter*s.

b. StandardSecurityManager

The *StandardSecurityManager* is an object that extends *Module*, allowing it to be a module within an NPSNET-V application, and implements the *SecurityManager* interface. This interface allows any object that implements the *SecurityManagerSubscriber* to create a communication link with the implementing object. This allows for the connectivity with the filters described in the next section.

The functionality of this object can be broken down into three areas: communications with the *SecureServer*, management of registered filters, and management of the distribution of received *SecretKeyPack*s destined for *SecureFilter*s.

c. Filters

Filter objects allow for the manipulation of data streams. These filters follow the filter pattern as described in section C.2 of this chapter. Each of the filters extends *ModuleContainer*, and implements the *Channel*, *ReceivedPacketListener*, *PropertyBearerListener*, and *SecurityManagerSubscriber* interfaces.

Filters perform operations on NPSNET-V *DataPacket* objects that pass through them. Each filter removes a byte array, representing the data, from the packet. The appropriate manipulation is performed on the byte array, and then a new *DataPacket* is generated with the new manipulated byte array, and sent along the communication stream.

The NSMS has three filter objects, each with unique functionality in manipulating the data contained within the *DataPackets*. These filters are: the *SequenceFilter*, the *IntegrityFilter*, and the *SecureFilter*. Section E of this chapter provides an in-depth overview of these objects.

4. Miscellaneous Components

The main components identified in the previous section require the use of several other objects. Some of these objects are basic to the NPSNET-V architecture, and allow the NSMS to function within it. Others were created to provide and assist with the functionality of the system. These are described in the below sections.

a. NPSNET-V Classes

Three NPSNET defined classes are used within the NSMS: *Module*, *ModuleContainer*, and *DataPacket*. *Module* is an abstract class that represents the base class of every NPSNET-V module. *ModuleContainer* is a class that allows for the containment of other *Modules* and *ModuleContainers*. An easy way to visualize this is to think of *Module* as only being able to be leaf nodes of a tree, and *ModuleContainers* as having the ability to be any kind of node in the tree.

The *DataPacket* object contains the entity state data to be transmitted across the network. It holds two primary data elements: an array of bytes representing the data, and an integer holding the length of the array.

b. Interfaces

Since interfaces are the primary means of communications between components within NPSNET-V, they play an important role in the NSMS. Two interfaces were designed to support the communication between the *StandardSecurityManager* and the filters.

SecurityManager: Implemented by the *StandardSecurityManager*, this interface was designed as a means for the filters to establish a connection with the *StandardSecurityManager*. Table 7 provides a listing of the methods required by this interface.

Returns	Method Name (parameters)	Description
void	addSecureSubscriber (*SecurityManagerSubscriber* sms)	Registers a filter with the *SecurityManager*
void	removeSecureSubscriber (*SecurityManagerSubscriber* sms)	Unregisters a filter with the *SecurityManager*

Table 7. *SecurityManager* Interface Methods

SecurityManagerSubscriber: Implemented by the filters, this interface was designed as a means for the *StandardSecurityManager* to communicate with the filters. Table 8 provides a listing of the methods required by this interface.

Returns	Method Name (parameters)	Description
void	beginPacketTransmission()	Signals filter to transmit packets
void	endPacketTRansmission()	Signals filter to stop transmitting packets
void	beginPacketReception()	Signals filter to receive packets
void	endPacketReception()	Signals filter to stop receiving packets
void	addKeyPack(*SecretKeyPack* skp)	Provides filter with a new *SecretKeyPack*
void	setApplicationID(long id)	Provides filter with the host applications ID
String	getID	Returns the filter's ID
SecretKeyPack	getCurrentKeyPack()	Returns the current *SecretKeyPack* in use, if any
Vector	getAllKeyPacks()	Returns all *SecretKeyPack*s that are waiting to begin their active period
int	getFilterType	Returns the filter's type (i.e. Secure, Integrity, Sequence)

Table 8. *SecurityManagerSubscriber* Interface Methods

These two interfaces provide the ability to create different types of *SecurityManager* and *SecurityManagerSubscriber* objects; they can be easily integrated as long as they contain the methods identified in the interfaces. This provides the ability to have several different types of *SecurityManager* type objects from which to choose depending on the security functionality that is desired.

The two interfaces that were developed as part of the NSMS were discussed above, but there are many other interfaces specific to NPSNET-V that were required to be implemented by NSMS objects. A brief description of each is presented:

Channel: This interface is used to identify objects used as entity state packet handlers and manipulators. It requires three methods be implemented: *sendPacket, addReceivedPacketListener, removeReceivedPacketListener*; the first

one is used for transmitting packets outbound; the last two are used for connecting the implementing object to connect to the next higher object in the application.

PropertyBearerListener: This interface is used for establishing channel-to-channel outbound communications. That is entity state packets from an application's entities are communicated down and out of the application through communication links established through this interface. It requires two methods be implemented: *propertybearerRegistered* and *propertyBearerDeregistered*. These are used to register and deregister objects that are interested in changes in the object with which they are registered.

ReceivedPacketListener: This interface is used for establishing channel-to-channel inbound communication. That is received packets from the network are communicated up the application through communication links established through this interface. It requires one method be implemented: *packetReceived*. This allows implementing objects to transmit received packets up the application to.

Startable: This interface is used for module control. It requires three methods be implemented: *start, stop, isRunning*; the first two are self describing, the third returns a Boolean indicating if the module is running.

A number of Java API interfaces were used as well. These include: *Runnable* and *Serializable*. Refer to [Flanagan99] for descriptions of these interfaces.

c. SecretKeyPack

The *SecretKeyPack* is a data-holding object that was created in order to facilitate the management and distribution of symmetric keys within the NSMS. It contains all the information needed by a *SecureFilter* for the proper operation of cryptographic operations. Table 9 identifies the contained data objects and their descriptions.

Data Item	Type	Description
filterID	String	The *SecureFilter* that this key is intended for
keyID	array of four bytes	The *SecretKeyPack*'s identifier in byte form
keyIDInt	integer	The *SecretKeyPack*'s identifier integer form
key	Key	The symmetric key object
initializationVector	Array of eight bytes	The initialization vector required for chaining
mode	String	The chaining mode to be used with the key
paddingScheme	String	The padding scheme to be used with the key
beginTime	long	The start time of the key's active period
endTime	long	The end time of the key's active period

Table 9. *SecretKeyPack* Data Items

d. SecureServerConnection

A *SecureServerConnection* object represent an SSL connection between the *SecureServer* and an individual *StandardSecurityMananger*. It is created by the *SecureServer* whenever a *StandardSecurityManager* successfully connects, receiving a handle to the SSL socket that was generated, an identifying integer value unique to that connection, and a handle back to the *SecureServer* to ensure two-way communications. Table 10 identifies the public methods of the *SecureServerConnection* object.

Returns	Method Name (parameters)	Description
Void	Run()	Threaded method that listens for communications from the *StandardSecurityManager*
Vector	getFilterVector	Returns a Vector containing the Filters associated with this connection
Void	sendKeyPack (*SecretKeyPack* keyPack)	Transmits the given *SecretKeyPack* to the StandardSecurityMananger associated with this connection.
Void	closeConnection()	Closes the connection's socket

Table 10. Methods of the *SecureServerConnection* object

e. KeyMaker

The *KeyMaker* object is instantiated by the *SecureServer*, and is used for symmetric key generation. It contains a *KeyGenerator* object that produces symmetric keys based on the algorithm that it is initialized with; currently three algorithms can be used: DES, DESede, and Blowfish.

The *KeyMaker* can be instantiated in three way. The default setting will initialize the *KeyGenerator* with the DES algorithm and identify the chaining mode and

PKCS5Padding (see [Oaks 01] and [Stallings99] for information on chaining and padding) as the relevant parameters to set within the *SecretKeyPack* for use by the ciphers engines in the *SecureFilter*s. The second way is by passing the key algorithm to the constructor; this will create a *KeyGenerator* with the passed algorithm, the remaining default settings will be applied to the *SecretKeyPack*. The third way is by passing the desired key algorithm, chaining mode, and padding scheme to the *KeyMaker*, overriding all the defaults. This functionality is provided for future use, currently only one chaining scheme is operable, as well as one padding scheme; these are currently set as the default as identified earlier.

A feature implemented in the *KeyMaker* is the ability to generate key with random key algorithm. When a flag is set by a call to the *randomAlgorthmOn* method, every successive key will be generated with a random algorithm (DES, DESede, Blowfish). This ability can be shut off by a call to the *randomAlgorithmOff* method. This capability is primarily of use only for testing purposes.

The public methods for the *KeyMaker* object are identified in Table 11. These methods provide for the required functionality of producing *SecretKeyPack*s for use with the *SecureFilter*s.

Returns	**Method Name (parameters)**	**Description**
void	changeKeyAlgorithm(String alg)	Initializes *KeyGenerator* with the new algorithm
void	changeCipherMode(String mod)	Changes the desired chaining mode to use with the key
void	changePaddingScheme(String mod)	Changes the desired padding scheme to use with the key
String	getKeyAlgorithm()	Returns the *KeyGenerator*'s current algorithm
String	getCipherMode()	Returns the current desired chaining mode
String	getPaddingScheme()	Returns the current desired padding scheme
SecretKeyPack	generateKeyPack (int keyID, long bTime, long eTime)	Returns a new *SecretKeyPack* with the passed KeyID, beginTime and endTime.
Void	randomAlgorithmON()	Sets the random key algorithm flag to true
Void	randomAlgorithmOff()	Sets the random key algorithm flag to false

Table 11. Methods of the *KeyMaker* Object

f. KeyStores & Certificates

A *KeyStore* is used to hold asymmetric private and public keys, and the associated certificate. A *TrustStore* contains trusted certificates that are used to validate the authenticity of presented certificates. In order to allow for the proper functioning of SSL sockets, both the *SecureServer* and the *StandardSecurityManager* are required to have associated *KeyStores* and *TrustStores*. [Oaks01] delineates the process by which to create a *KeyStore* and generate a private/public key Pair. [Oaks01] delineates the process of generating a certificate corresponding to a key pair, and importing the certificate into an appropriate TrustStore.

In order for an SSL socket to function, both the client and the server needed to have an *SSLContext* instantiated, that were initialized with the locations of the relevant KeyStores and TrustStores that the connections would use. *The KeyStores* were named: *.serverKeyStore* and *.clientKeyStore*. The TrustStores were named *.serverTrustStore* and *.clientTrustStore*. These stores were placed in a folder called *TestStores*, and placed in the root directory (c:\). Table 12 displays the described directory structure and relationships.

Object	**KeyStore**	**TrustStore**
SecureServer	C:\TestStores\.*serverKeyStore*	C:\TestStores\.*serverTrustStore*
StandardSecurityManager	C:\TestStores\.*clientKeyStore*	C:\TestStores\.*clientTrustStore*

Table 12. KeyStore and TrustStore Locations

D. COMMUNICATIONS

The three main objects of the NSMS required various methods of communications. This section describes the different communication links used.

1. SecureServer – StandardSecurityManager

Communication between the *StandardSecurityManager* and the *SecureServer* was performed through SSL connections. Standard SSL protocol has only the server authenticating itself to the client. However, in order to provide authentication of the *StandardSecurityManager* component to the *SecureServer*, the server's *SSLServerSocket* was configured to ensure that the client authenticates itself to the server.

In order for the *StandardSecurityManager* to contact the *SecureServer*, it must hold the valid IP address for the server. This parameter is hard coded into the *getServerConnection* method of the *StandardSecurityManager* with the following statement: '*addr = InetAddress.getByName("131.120.7.142");*', where 'addr' is an *InetAddres*s object. The server is also hardwired to port 9096; this is identified in the *StandardSecurityManager* by *serverPort = 9096;* in the variable declaration section.

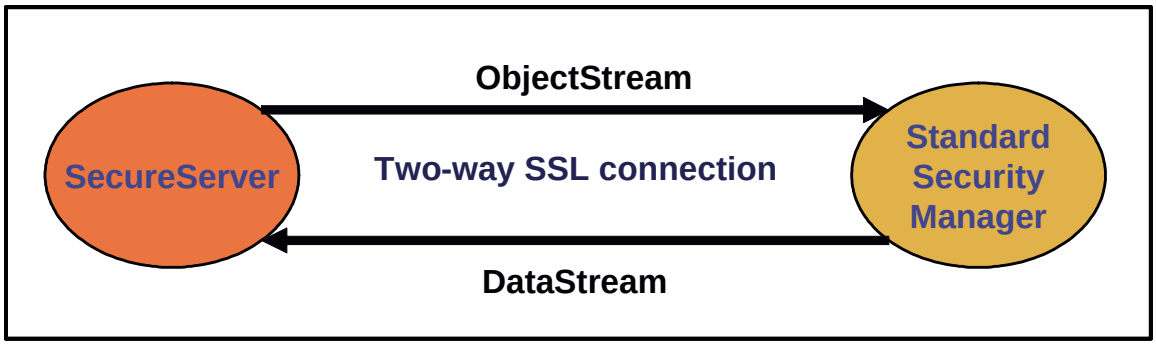

Figure 4. SecureServer / StaandardSecurityManager connection

Once a successful SSL handshake is completed, the server creates a *SecureServerConnection* object that contains the SSL connection with the *StandardSecurityManager*. All further communications occur through this object.

With a *SecureServerConnection* established, the server's only communication to the SecurityManger is the transmission of *SecretKeyPack* objects. This lent itself to the use of Java's Serialization interface, which converts objects into a series of bytes that can then be reconstituted into an object.

Transmissions from the *SecurityManager* to the *SecureServer* encompassed passing an application's ID to the server; and then, whenever a *SecureFilter* was created, registering that filter with the server. Once a filter is registered, the server transmits the current *SecretKeyPack* that was assigned for that filter's ID; if that ID had not been registered yet, a new *SecretKeyPack* would be generated and sent.

2. StandardSecurityManager – Filters

The main communications between the *SecurityManager* and filters are performed through the use of *SecurityManager* and *SecurityManagerSubscriber* interfaces as described earlier. Figure 5 graphically depicts this relationship.

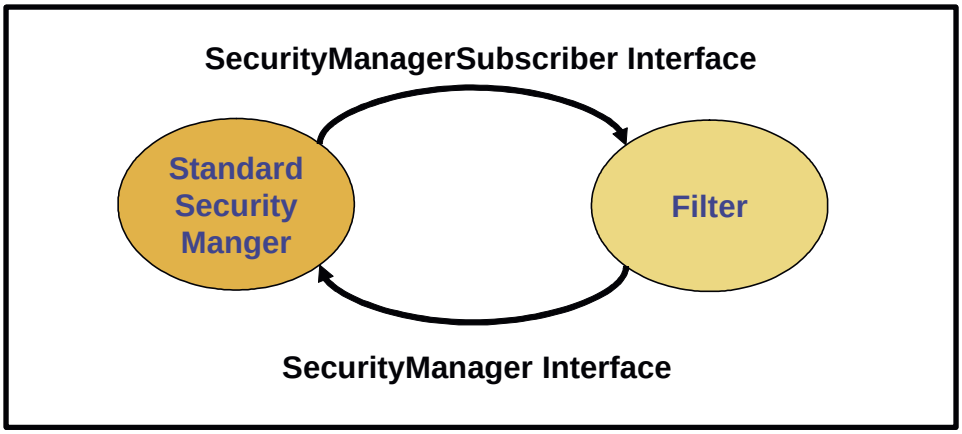

Figure 5. Communication Interfaces Between StandardSecurityManager and the filters

3. Filter – Filter

The main communications between the individual filters, and between the filters and the channel and NetworkController objects are performed through the use of PropertyBearerListener and ReceivedPacketListener interfaces as described earlier. Figure 6 shows this relationship.

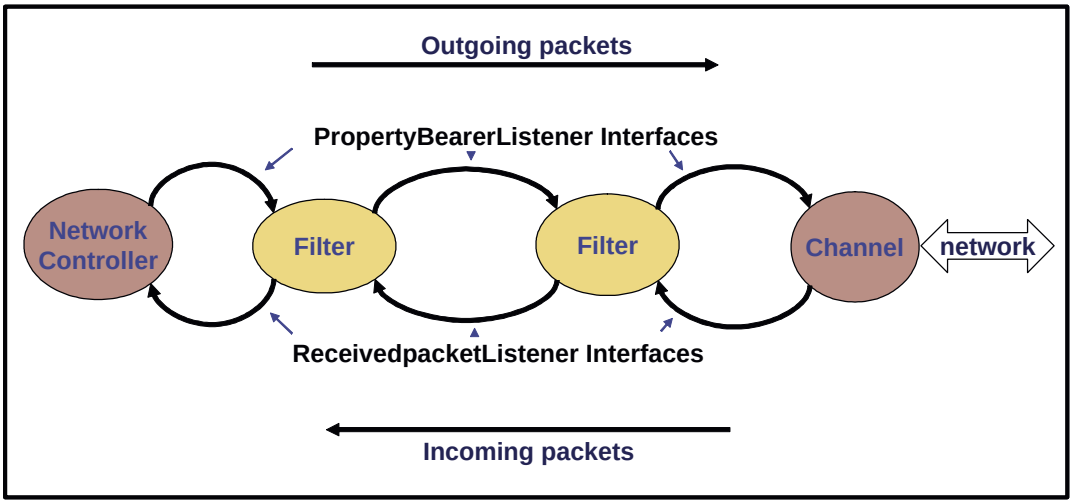

Figure 6. Communication Interfaces between filters

Figure 6 shows how an outbound packet travels from the *NetworkController* through the connections established through the PropertyBearerListener interfaces out to the network. Incoming packets likewise are received from the network by a channel object; the packets then travel up the application through the connections established by the ReceivedPacketListener interface. Each filter in this sequence performs their specific operations on the data passing through it, and then passes the new data to the next filter.

E. FILTERS

These filters perform unique manipulations on the data contained in a DataPacket. The data is in the form of byte arrays of arbitrary length. Each filter removes the data array, manipulates it as necessary, and then generates a new DataPacket with the new data array, and passes it off to the next module, be it a NetworkController, filter, or channel object.

1. SequenceFilter

This filter performs sequencing operations on the DataPackets. The purpose behind this is to avoid replay attacks in which an attacker listening on the network makes copies of legal packets and then resends the packets, in the hopes that the packets will be accepted as legitimate. Since the packets were originally encrypted with a legitimate key, and the message digest code is correct, detecting replayed packets is a non-trivial problem. If this technique is not countered, attackers can force certain events to re-occur at will. Typically this type of attack is countered by adding unique information to each packet so that duplicate packets can be detected and rejected. Appendix D contains the code for this filter.

When first instantiated, this filter selects a random, four-byte integer as the beginning sequence number. For outbound packets, it appends the application's eight-byte ID and a four-byte sequencing number into the beginning of the data array, thus creating a new byte array that is twelve bytes larger than the original; Figure 7 depicts this operation. It then creates a new DataPacket with this data array and passes it to the next filter or channel. The sequence number is incremented after each packet. When the sequence number reaches the maximum integer value of 2^{31}-1, it rolls the sequence number over to begin at the lowest integer value of -2^{31}, allowing for continued sequencing support.

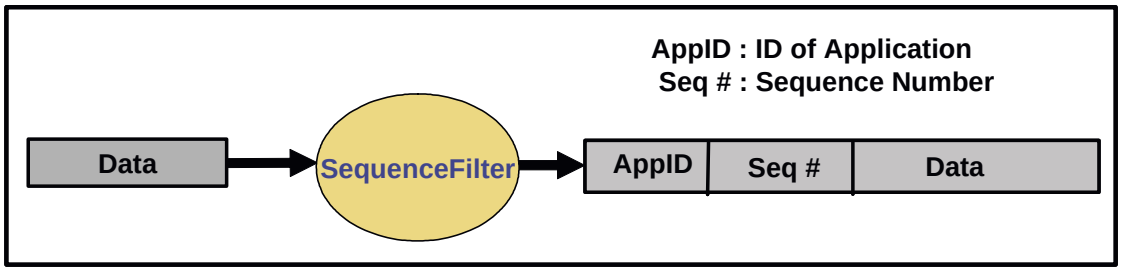

Figure 7. Outbound Data Before and After the SequenceFilter

On incoming DataPackets, this filter performs sequence number verification, using the *appSequenceTable*, a *HashTable* object, called in which it maintains known application IDs and the most recent sequence number observed for that ID. When it receives an inbound packet, it retrieves the data array and removes the application ID and the sequence number. It sends these two items through a *checkIDandSequence* method for verification.

The *checkIDandSequence* method first checks to see that the application ID is not the same as the host application ID; if it is the same, the method signals for the packet to be rejected. Next, the method checks the *appSequenceTable* for the presence of that particular ID; if the ID is present then the sequence number is checked against the last seen sequence number to ensure that it is within twenty increments one of the last packet received from that application; this is to take into account the possible packet loss when dealing with UDP connections. If the number is legitimate, then the sequence number is replaced in the table and the method signals that the packet is good. If the application ID is not present, then the filter accepts it as a legitimate ID, and adds it and its sequence number to the *appSequenceTable*, then signals to accept the packet. If the DataPacket is accepted, then the filter creates a new DataPacket with the data portion of the data array, and passes it to the next filter or network controller.

2. IntegrityFilter

This filter performs integrity operations on the DataPackets. The purpose behind this is to ensure that the data transmitted in the data array has not been altered, either through inadvertent corruption or malicious attack. The integrity feature is provided

through the use of a *MessageDigest* object from the Java Security API. Refer to appendix E for this filter's code. This filter uses two *MessageDigest* objects: *transmitMessageDigester* and *messageDigestChecker*.

For outbound packets, this filter takes the data array and provides it to the *transmitMessageDigester*. It then calls the objects *digest* method, which produces the message digest as an array of bytes. The MessageDigest objects in the filter use the Secure Hash Algorithm (SHA), which produces a twenty byte message digest on any length of byte array that is provided. In anticipation of future functionality that may produce variable sized message digests, the length of the digest is then determined. The length of the digest, as a one-byte integer, and the digest are then appended to the beginning of the data array (see Figure 8) and a new DataPacket is created with the new data array. The packet is then delivered to the next outbound filter or channel. As this filter currently functions, it increases the size of any original array by twenty-one bytes.

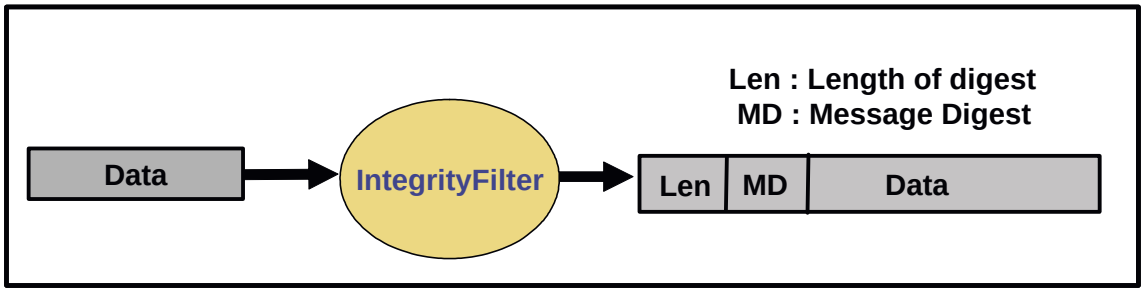

Figure 8. Outbound Data Before and After the IntegrityFilter

On incoming DataPackets, this filter performs message digest verification. When it receives an inbound packet, it retrieves the data array and removes the one-byte length, and subsequently the message digest. It then provides the message digest and the remaining data array to the *verifyDigest* method for verification processing.

The *verifyDigest* method uses the *messageDigestChecker* to produce another message digest on the provided data array. This new digest is compared to the provided digest. If the digests are identical, then the method signals that the data has not been modified and is okay to continue. If the data is verified to be unmodified, then the filter

creates a new DataPacket with the data portion of the data array, and sends it up the application.

3. SecureFilter

This filter performs cryptographic operations on the DataPackets. This addresses the communication confidentiality of the DataPackets. Cryptographic operations are conducted using the *Cipher* object from the JCE extension of the Java Security API. The filter contains two of these *Cipher* objects: *encipherer* and *decipherer*. Details on the initialization of these ciphers are discussed in section G.3.b. Refer to appendix F for this filter's implementation.

For outbound packets, this filter takes the data array and provides it to the *encipherData* method. Within this method the data array is passed to the *encipherer* and the data is encrypted by providing its *doFinal* method. A new byte array is then formed with the current key's four-byte ID and the encrypted data array, which will have increased in size to be a multiple of eight bytes; this is due to the requirement for ciphering in sixty-four bit blocks by the cipher in chaining mode. The new data array is then returned. A new DataPacket is formed with the array, and then sent to the next outbound filter or channel, as shown in Figure 9. The current implementation of this filter increases the size of the original array by four to eleven bytes.

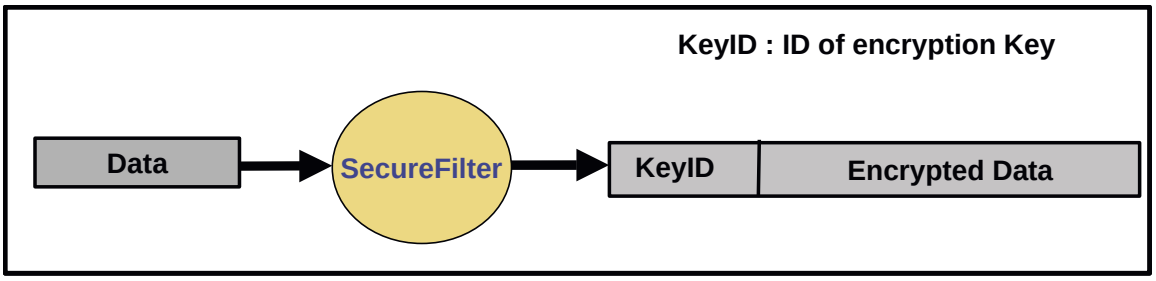

Figure 9. Outbound Data Before and After the SecureFilter

The filter performs decryption operations on incoming DataPackets. When the filter receives an inbound packet, it retrieves the data array and provides it to the *decipherData* method. Here, the key ID is separated from the encrypted data array. The key ID is then checked to ensure it is the same as the filter's current active key; if not, an exception is raised. Otherwise, the encrypted data is passed to the *decipherer*, and its

doFinal method is called, producing the deciphered byte array. This new data array is then returned. Next, the filter then creates a new DataPacket with the data array, and sends it up the application.

This filter is the most complicated of all the filters designed in the NSMS so far. There is an infrastructure used for managing the *SecretKeyPack*s within the *SecureFilter*. This infrastructure is described in section G.3.

F. MODULE MANAGEMENT

In theory, an infinite number of participants could exist simultaneously in an RTEVE. Here we discuss how the SecurityManagers and the filters would be used to support large numbers of participants.

1. Management of SecurityManagers

The *SecureServer* must be able to manage any number of *StandardSecurityManager* connections. This is accomplished through the use of a unique identifier for every *StandardSecurityManager*, and through the *SecureServer*'s use of an efficient data structure to track the *StandardSecurityManager* connections.

a. Application ID

When a *StandardSecurityManager* is first instantiated, it determines a unique eight-byte identifier by concatenating a series of four random bytes to the four–byte IP address; it then determines the long integer that this eight-byte sequence represents. This long is then identified as the *applicationLong*. This is designed to avoid collision of identifiers when several applications are sharing IP addresses. There does exist the possibility of collision, but it is deemed a very remote chance when dealing with four random bytes.

b. SecureServer's Role

The *SecureServer* manages the *StandardSecurityManager*s, through the use of a Vector that contains each *SecureServerConnection* object, one *SecureServerConnection* per *StandardSecurityManager*. If a socket connection is dropped, the *SecureServer* handles the exception by removing the *SecureServerConnection* object and deregistering any filters that were attributed to that *SecureServerConnection*.

2. Management of Filters

The management of the filters is crucial to the security of the application, particularly with the *SecureFilter*.

a. Filter ID and Type

Since it is possible to have hundreds of filters within the same application, every filter must be distinguishable: distinguishable from other types of filters, and distinguishable from others of the same type. In order to identify different types of filters, the *SecurityManagerSubscriber* interface contains three constants that are used for filter identification: SECURE_FILTER_TYPE, SEQUENCE_FILTER_TYPE, and INTEGRITY_FILTER_TYPE. This method is used in order to provide a common base for all modules to be able to identify types.

In order to distinguish a filter from others of the same type, a unique string must be passed into the filter during instantiation. This is performed by adding a line to the XML configuration file to call the *setID* method of the filter, and providing an identifying name.

b. StandardSecurityManager's Role

The *StandardSecurityManager* maintains three *Vector* objects, one for each type of filter. When a filter registers itself with *StandardSecurityManager*, the *SecurityManager* retrieves the filter's type and ID, adding the filter to the appropriate vector. If the filter is a *SecureFilter* object, the *StandardSecurityManager* then registers the filter with the *SecureServer*.

When the *StandardSecurityManager* receives a *SecretKeyPack* from the *SecureServer* for a filter it contains, it immediately hands the *SecretKeyPack* to the *SecureFilter* for processing.

c. SecureServer's Role

The *SecureServer* maintains two HashTables that it uses in managing *SecureFilter*s. The filterConnectionTable maintains a list of all active *SecureFilter*s that have been registered with the server, along with a vector that contains every *SecureServerConnection* object that holds that particular filter. This is used to efficiently identify what *SecureServerConnection*s need to receive a *SecretKeyPack* that is intended for use by a particular filter.

The *FilterKeyTable* is used to efficiently map an active *SecureFilter* with its currently active *SecretKeyPack*. This way, when a *SecureServerConnection* registers a filter that already exists, it will immediately receive the *SecretKeyPack* that is in use by the other *SecureServerConnection*s with the same filter, and the filter can immediately begin to communicate with its clones in other applications.

G. KEY MANAGEMENT

Key management is fairly straightforward within the NSMS. It begins with the generation of keys by the *SecureServer*, and ends with the *SecureFilter* managing the key changes. The central object within this area is the *SecretKeyPack*, presented in section

1. SecureServer

The *SecureServer* is at the heart of the key management infrastructure of the NSMS. It handles generation, distribution and tracking of all keys for all of the *SecureFilter*s.

a. Key Generation

The *SecureServer* uses the *KeyMaker* to generate needed *SecretKeyPack*s. For this implementation of the NSMS, a graphical user's interface, the *SecureServerGUI*, was created testing the key-generation functions of the system. It contains buttons that allow for generation of a *SecretKeyPack* destined for all *SecureFilter*s. In addition it contains a button that initiates a continuous generation of *SecretKeyPack*s at an interval identified in the server. It contains a button that switches the random key algorithm capability on and off. Also, whenever an individual *SecureFilter* is registered, a button is added to the GUI that allows for a *SecretKeyPack* to be generated and sent just to that filter.

Upon request for a new *SecretKeyPack*, the *KeyMaker* generates a new key by using a *KeyGenerator* object that is set to the currently selected algorithm (DES, DESede, or Blowfish). It then generates a random array of eight bytes for use as the initialization vector that will be used by the ciphers in the *SecureFilters* for the purposes of chaining. The *KeyMaker* then instantiates a *SecretKeyPack* that contains the key, the initialization vector, and all other information necessary for the proper handling of the key and functioning of the ciphers; refer to section C.4.c for details on the *SecretKeyPack* data.

b. Key Distribution

The *SecureServer* handles the distribution of keys through the use of the *filterConnectionTable*. When a *SecretKeyPack* is generated for a particular filter, it runs through the *SecureServerConnections* contained in the vector associated with the filter and sends the *SecretkeyPack* to each one.

c. Key Tracking

Key tracking is handled through the use of the *filterKeyTable*. This table contains the active *SecureFilter* IDs and their corresponding active *SecretKeyPack*s.

2. StandardSecurityManager

The *StandardSecurityManager* perform minimal actions with the *SecretKeyPack*s. When one arrives from the *SecureServer*, it looks at the ID of the filter it is intended for, and passes it to the filter. No tracking is performed by the *StandardSecurityManager*. Since the filter will maintain the *SecretKeyPack*s, there is no need for the *StandardSecurityManager* object to do so as well.

3. Secure Filter

The *SecureFilter* performs the detailed key management operations. It handles immediate key changes, and schedules and executes future key changes based on the indicated active period of the received KeyPacks.

a. Key Tracking

The *SecretKeyPack* that is currently active is identified as the *currentKeyPack*, all cryptographic operations on outbound and inbound DataPackets are performed by referencing the data in this keypack. All *SecretKeyPack*s that are received for future activation are placed in the *nextKeyPacks* Vector in order of its start time.

b. Key Changing

The *SecureFilter* performs a key change by shutting down the filter's transmission and reception capabilities, instantiating two new *Cipher* objects with the key parameter information contained in the SecretKeyPack. These two ciphers, the *encipherer* and *decipherer*, are then initialized with the initialization vector and the new key provided by the *SecretKeyPack*. The filter's transmission and reception flows are then restarted.

If a key pack is received with a start time that is in the past, the *SecureFilter* performs an immediate key-change operation in which all awaiting key packs are removed from the Vector, and a key change is performed. When the active period has not yet begun, a *TimerTask* object, that includes the *changeKeys* method of the filter, is created. This task is then handed to a *Timer* object, along with the delay before the active period begins. The *Timer* ensures that the TimerTask is begun at the appropriate time. When the *TimerTask* is tripped, the appropriate key change is performed.

H. NSMS XML CONFIGURATION FILES

An XML configuration file for a sample NSMS containing world is presented in Appendix C. The application represented in this configuration file contains a *StandardSecurityManager* and four filters. One Secure filter, that is assigned an ID of 'sec96' and is placed by itself before a multicast channel that transmit/receives on address 225.93.23.96. The remaining three filters are placed in series before a multicast channel transmitting/receiving on address 225.93.23.92. The three filters, in outbound order are: *SequenceFilter* 'seq92', *IntegrityFilter* 'int92', and *SecureFilter* 'sec92'. This indicates that a *DataPacket* will first be processed by the *SequenceFilter*, then that packet will be processed by the *IntegrityFilter*, and finally that packet will be processed by the *SecureFilter* before being transmitted over the multicast channel. The XML files used for testing purposes are identified in Table 13, along with the implemented filters and their IDs.

XML File	contained filters	Filter ID
secure_server.xml	n/a	n/a
secure_a.xml	SecureFilter	sec96
secure_b.xml	SequenceFilter	seq92
	IntegrityFilter	int92
	SecureFilter	sec92
	SecureFilter	sec96
secure_c.xml	SequenceFilter	seq92
	IntegrityFilter	int92
	SecureFilter	sec92
	SequenceFilter	seq93
secure_d.xml	SequenceFilter	seq92
	IntegrityFilter	int92
	SecureFilter	sec92
	IntegrityFilter	int 95
secure_e.xml	IntegrityFilter	int95
secure_f.xml	SequenceFilter	seq93

All xml files are located in: npsnetv\applications\tests
The last two digits of a filterID equate to a Multicast channel

Table 13. NSMS test XML Configuration Files

I. NSMS WEAKNESSES

There are several weaknesses that have been identified with the NSMS as it currently is implemented. These are:

- The centralized server architecture. This poses a single-point-of-failure vulnerability for the system. Also, if the server were to be subverted, then the subverting entity would have complete control of the interaction capabilities of the system.
- The keys that are held by the *SecureServer* and the *SecureFilter*s are vulnerable as they reside in memory. If a malicious entity had access to the system, and could determine where the keys resided in memory, the malicious entity might try to reconstitute the keys to gain access to the data packet transmissions.
- The KeyStores contain the public/private key pairs and certificates. If copies of these are acquired by a malicious entity, then that entity could use the key pairs or certificates for authentication purposes and potentially gain access to the worlds.

- The *SequenceFilter* currently will trust any new application ID that it sees, without authenticating it. If a malicious entity were able to inject packets without registering with the *SecureServer*, the *SequenceFilter* would trust them and accept them as a valid user. Authentication is assumed through the use of a *SecureFilter* combined with an *IntegrityFilter*(i.e., if the incoming packet has an active key, and the packet decrypts correctly, and the packet has a valid message digest, then it is presumed to have originated from an authenticated application that possesses valid encryption keys). This requires the use of a *SecureFilter* with the *IntegrityFilter*. The *SequenceFilter* should be able to authenticate on its own, possibly by verifying the new application ID with the *SecureServer*.

- The *IntegrityFilter* currently uses a MessageDigest, which, if not used in conjunction with the *SecureFilter*, would be vulnerable to attacks. If the *IntegrityFilter* were to be used alone, then a MAC that requires the use of shared keys to generate the digests might be more appropriate to use. This would still maintain the authentication aspect due to the shared secret key that was used. Public key encryption could also be used to encrypt and digitally sign the message digest and provide a layer of security for the integrity operations, as well as provide non-repudiation characteristics. However, this possesses two major drawbacks for state data packet transmissions: It would require a complicated public key distribution capability; and, more importantly, the public key encryption algorithms would induce too great a delay in packet processing.

J. KEY DISTRIBUTION LATENCY PROBLEM

Synchronized key changes amongst all *StandardSecurityManager*s would be a non-issue if packet distribution from the *SecureServer* were instantaneous. However, the issue of latency is a challenge when dealing with this issue. Figure 10 depicts the latencies that are present in the key distribution problem.

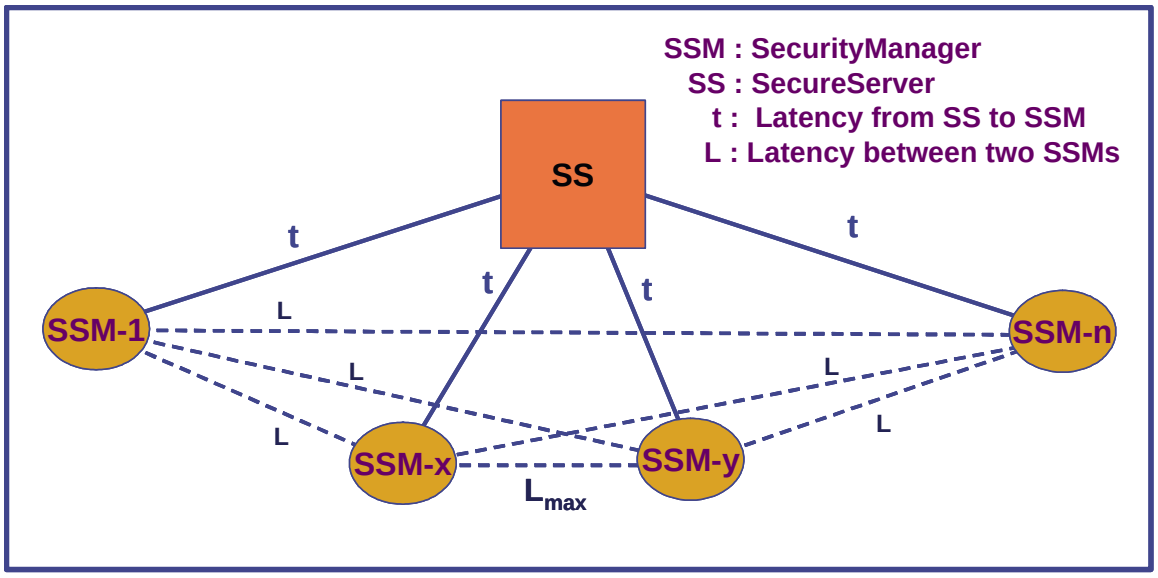

Figure 10. Delay diagram of NSMS architecture

One impact on the performance of the system is the time it takes for a key to be distributed to all participants. If there were *n* participants and the time it takes for a key to be transmitted to a participant is *t*, then the total time required for all participants to receive the new key would be *(n*t)*. Since a disparity in the keys held by the participants would occur when the first participant received their key, then the time between the first and last participants receiving their keys would be ((*n*-1)**t*). If the key on the hosts is changed immediately upon receipt, this means that during this period of time the entire set of hosts will not have a consistent shared key. Thus the messages of some hosts will be unintelligible to others. This problem can be minimized by coordinating a time at which the switchover will occur. The *SecureServer* sends out a new key along with a time that all hosts will switch to the new key. The hosts (which are assumed to have synchronized clock times) all switch to the new key at the same time. The coordinated switchover approach works well for routine shared key changes. However, if we suspect the key is compromised, or if one host leaves the VE and we wish to minimize the amount of data that is transmitted with a compromised key, we may accept a period of mutual unintelligibility in order maximize data security.

The second aspect of this problem is the latency between hosts in the VE. If the longest latency between any two participants were L_{max} then it would take L_{max} time

before one of those participants would see a packet encrypted with the new key from the other participant. Assuming that all hosts switch to a new key at the same time, the packets already in the network that were encrypted with the old key would still be in transit, and therefore arriving over a period of L_{max}.

The two above mentioned aspects must be taken into account by a SecureServer when generating a new key, in order to ensure minimal impact due to mismatched keys amongst participants. This theoretical minimum time delay between the transmittal of a key to the first host, and the beginning of the key's active period can be computed to as: $(n\text{-}1)*t + L_{max}$. This requires that the *SecureServe*r be aware of the performance within the network, in order to accurately identify the network's latency.

K. SUMMARY

In this chapter we gave an overview of desirable characteristics and requirements that a comprehensive security management system for an RTEVE should possess. Our NSMS is not an all-encompassing solution. Rather, it is the beginning of a more comprehensive approach for addressing issues of security in RTEVEs.

Figure 11 depicts an NSMS in a two-application environment in which two channels of communication are being used. One channel is a TCP/IP connection that has a *SecureFilter* encrypting the data that is transmitted. The second communication channel is a UDP multicast channel in which the data packets are sent through a *SequentialFilter*, then an *IntegrityFilter*, and finally through a *SecureFilter* (the recommended sequence of filtration).

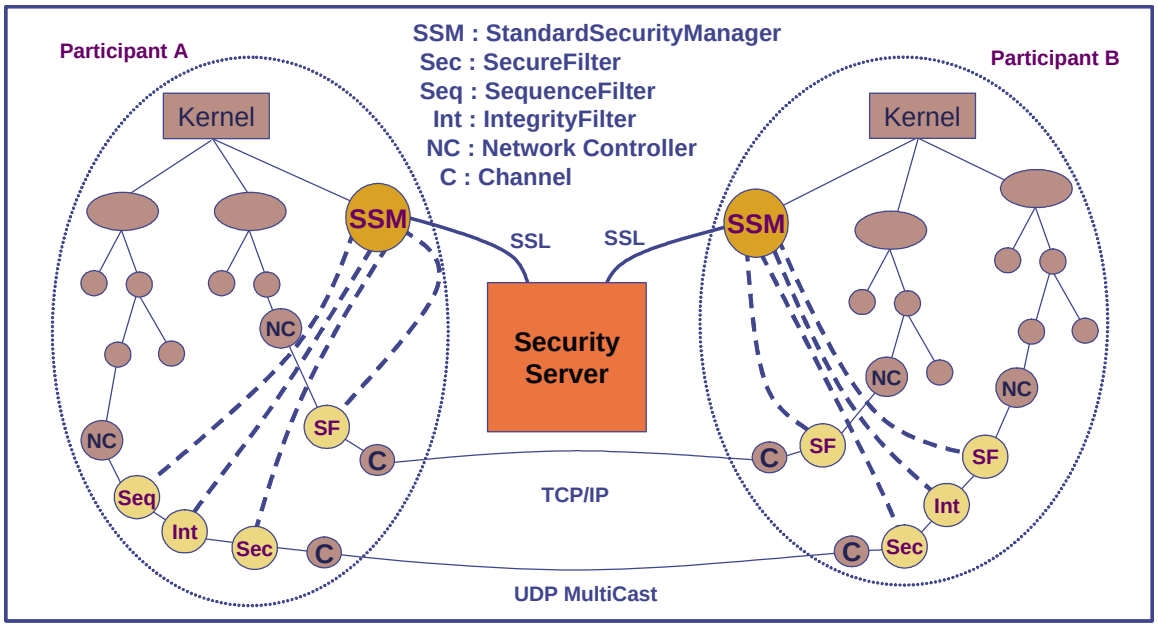

Figure 11. NSMS in a Two Application Environment

The next chapter documents the results from several studies performed on the filters that were designed. The studies were conducted to identify the impact of the filter algorithms on process-induced delay in packet transmission.

THIS PAGE INTENTIONALLY LEFT BLANK

V. PERFORMANCE REVIEW OF NSMS CAPABILITIES

This chapter presents a beginning analysis of the NSMS filters with respect to the amount of delay induced by their algorithms. This provides insight into the effects of these security mechanisms on data-packet-flow performance of a large scale VE. Since reliability is more of a system-level concern, we choose to focus on investigating delay and bandwidth.

A. INTRODUCTION

There has always been a concern with the impact of security features on the performance of a networked VE. These studies were designed to identify the impact that the security-enabling filters introduced in this thesis would have on the QOS concerns identified in chapter II.

B. SYSTEM SET-UP

Since the studies were focused on the performance of the application's filters, it was necessary to ensure that the application would have its own dedicated host; otherwise, performance would be impacted by the presence of the server on the same computing platform. Therefore a dual-platform configuration was devised with the SecureServer and an individual NPSNET-V application operating on separate computers.

1. Server

The server was hosted on an IBM ThinkPad iSeries 600 MHz Celeron laptop, with 192MB RAM and the Windows2000 operating system. It was connected to the local area network through standard 100 mbit/sec Ethernet cabling.

2. Experiment Applications

The NPSNET-V applications were hosted on a Dell Dimension 4100 Intel P-III 1GHz desktop, with 256MB RAM, and the Windows2000 operating system. This too was connected to the local area network through standard Ethernet cabling.

Base performance characteristics of the filters were of interest. Each experimental VE application was designed with only one entity in order to produce a stream of constant-size data packets. In order to avoid the impact of computation by graphical processes, the 3-dimensional viewing components were not included in the test applications. The only processes executing, other than the study application, were the

primary process threads of Windows2000 running in the background; this ensured that the application under study would be the only main draw on processor resources.

C. GENERAL STUDY DESIGN

In order to gather time statistics within the filter, Java's *System.currentTimeMillis* method was used to retrieve the current time in milliseconds as a long value. Since most processing was believed to be on the sub-millisecond level, this would present precision errors. In order to alleviate this concern, a decision was made to encase the functional areas of each filter in a *for*-loop that would loop through the encased algorithm for 10000 iterations. The system time was recorded on entering and exiting the loop. The difference between the times was then divided by the number of loop iterations, with the resulting value being attributed to one cycle of the algorithm. This *for*-loop was then executed thirty times and averages determined to produce one data point, with each data point representing thirty runs of 10000 iterations. Ten data points were then accumulated for each parameter set of the individual studies and analysis performed.

We compared the execution times of the algorithms by the size of the data arrays. In order to produce this effect, each study used two different NPSNET-V applications, each containing a different entity that produced different sized packets. The generated packets had byte arrays with lengths of 116 and 156 bytes. These packet sizes are fairly typical for many VE network protocols, such as DIS. Packets with data of length 116 bytes were produced by the *StandardExplosionManager* entity, while data packets of 156 were produced by the *teapot* entity.

Due to the number of iterations for each sample, precision of the execution times was limited to three decimal points. One thousandth of a millisecond precision was deemed sufficient for this study. Therefore, where there seems to be no difference in execution comparisons, there may in fact be a small difference, albeit minute.

All studies include results on both the outbound and inbound packet handling algorithms. Separate *for*-loops were used on each algorithm within the same filter, and were performed in alternating order using the same data. That is, one packet was run through the outbound process, and then that resultant packet was processed through the

inbound algorithm; the same cycle was repeated thirty times to produce one data point on each algorithm.

D. SECURE FILTER DELAY STUDY

This study delved into the effects of the enciphering and deciphering capabilities of the filter. Concern has always been expressed that the delay induced by these processes would negatively impact QOS, and thus is usually not considered with respect to entity data packets.

1. Study Design

This study was designed to look at the impact of the actual *SecureFilter*'s enciphering and deciphering algorithms, and that introduced by the actual ciphering and deciphering of the data when using the three different keying algorithms of DES, DESede, and Blowfish (Blowfish was set to use the maximum key size of 448 bits). Note that the data sizes in the charts of this section indicate the size of the data arrays that are being passed into the actual cipher/deciphering portion of the algorithm; in the case of 156/160, 156 bytes are entering the enciphering Cipher, while 160 bytes are entering the deciphering cipher. This is caused by padding in the encryption algorithm that generates ciphered data in chunks of eight bytes. One hundred fifty six bytes are padded out to the next multiple of eight bytes, 160 bytes. The deciphering algorithm therefore receives 160 bytes and produces deciphered data of 156 bytes.

The first two experiments focused on the execution times of the *SecureFilter*'s algorithms and the actual cipher/decipher calls respectively. The third experiment focused on the impact due to different key sizes when using the Blowfish algorithm; and the last experiment focused on the processing impact caused by the *SecureFilte*r's operation during the execution of the encryption operations.

2. Results

Analysis of Enciphering/Deciphering Algorithm Execution Time: This experiment was designed to provide data on the filter's ciphering and deciphering algorithm for data array sizes of 156 and 116 bytes. It was performed using the three identified keying algorithms. Results of this experiment are shown in Table 14, and indicate that:

- Overall execution times of the algorithms increase as you go from Blowfish to DES, and then to DESede for both enciphering and deciphering. The data indicates that Blowfish is approximately 0.017 milliseconds faster than DES, and 0.092 milliseconds faster than DESede for enciphering, and 0.013/0.089 for deciphering.
- Deciphering operations are less than 0.01 milliseconds slower than enciphering operations.
- Blowfish, considered the strongest of the three algorithms to break, was the fastest of all, even though it was set with the largest key size of all.

SecureFilter Algorithm (milliseconds)						
data size: 156/172 bytes				data size: 116/132 bytes		
Algorthm	Encipher	Decipher		Algorithm	Encipher	Decipher
DES	0.048	0.053		DES	0.036	0.040
DESede	0.123	0.127		DESede	0.092	0.096
Blowfish	0.031	0.035		Blowfish	0.024	0.026

Table 14. Average Execution Times for *SecureFilter* Encipher/Deciphering Algorithms per Key Algorithm

A comparison of the algorithms based on differing data array sizes is provided in Table 15. This comparison indicates the following:

- The execution time difference between enciphering and deciphering is virtually the same for each key algorithm, with the largest disparity residing with the Blowfish algorithm.
- A data size difference of forty bytes has an impact on the algorithms that is measurable only in the hundredth-of-a-millisecond range, a negligible difference.

SecureFilter Algorithm (milliseconds)			
Algorithm	Data size	Encipher	Decipher
DES	156/160 bytes	0.048	0.053
	116/120 bytes	0.036	0.040
	difference:	0.012	0.013
DESede	156/160 bytes	0.123	0.127
	116/120 bytes	0.092	0.096
	difference:	0.031	0.031
Blowfish	156/160 bytes	0.031	0.035
	116/120 bytes	0.024	0.026
	difference:	0.007	0.009

Table 15. Comparison of the *SecureFilter*'s Encipher/Decipher Algorithm Execution Times in Relation to Data Array Size

Analysis of Cipher's Encipher/Decipher Execution Times: This experiment was designed to provide data on the *Cipher* object's ciphering and deciphering process for data array sizes of 156 and 116. This information, compared to the data from experiment one, provided an indication of what delay was introduced by the non-cipher portions of the filter algorithms. Results of this experiment are shown in Table 16, indicating the following:

- Ciphering and deciphering operations using DESede are 200% slower than when using DES.
- Ciphering and deciphering operations using Blowfish are 50% faster than when using DES.
- There is negligible difference (in the thousandths of a millisecond) in execution times for ciphering/deciphering with the same algorithm.

Cipher Call (milliseconds)					
data size: 156/172 bytes				data size: 116/132 bytes	
Algorthm	Encipher	Decipher		Algorithm	Encipher	Decipher
DES	0.046	0.050		DES	0.034	0.037
DESede	0.120	0.123		DESede	0.090	0.093
Blowfish	0.028	0.032		Blowfish	0.022	0.023

Table 16. Average Execution Times for Cipher Encipher/Decipher Call per Key Algorithm

A comparison of the Cipher enciphering/deciphering calls on differing data array sizes is provided in Table 17. These results indicate that the Blowfish, DES, and DESede enciphering execution times are affected by differences in data array sizes. The affect was approximately 0.007, 0.011 and 0.03 milliseconds slower for an array difference of forty bytes.

Cipher call comparisons (milliseconds)			
Algorithm	Data size	Encipher	Decipher
DES	156/160 bytes	0.046	0.050
	116/120 bytes	0.034	0.037
	difference:	0.012	0.013
DESede	156/160 bytes	0.120	0.123
	116/120 bytes	0.090	0.093
	difference:	0.030	0.030
Blowfish	156/160 bytes	0.028	0.032
	116/120 bytes	0.022	0.023
	difference:	0.006	0.009

Table 17. Comparison of the Cipher Object's Encipher/Decipher Execution Times in Relation to Data Array Size

The difference between the algorithms and the cipher execution times is shown in Table 18. These results indicate that the average execution time of the filter per encipher and decipher that is attributable to the non-cipher algorithm is less than 0.003 milliseconds. Thus the performance of the filter rests predominantly with the key algorithm in use.

Difference Between Algorithm and Cipher Call (milliseconds)						
data size: 156/172 bytes				data size: 116/132 bytes		
Algorthm	Encipher	Decipher		Algorithm	Encipher	Decipher
DES	0.002	0.004		DES	0.002	0.003
DESede	0.003	0.004		DESede	0.002	0.003
Blowfish	0.003	0.003		Blowfish	0.002	0.003
Average:	0.003					

Table 18. Differences Between Method Call and Cipher call

Analysis of the Blowfish Algorithm Encipher/Decipher Execution Times: This experiment was designed to determine the effect of different key sizes on the encipher/decipher execution times of the Blowfish key algorithm. The three key sizes used were: 56, 128, and 448 bits. Results of this experiment are shown in Table 19, and indicate that key size appears to have a negligible influence, of less than 0.001 milliseconds, on enciphering and deciphering times. There is no reason to use less than 448-bit encryption since it has no adverse impact.

Blowfish cipher times (milliseconds)		
KeySize	Encipher	Decipher
56 bits	0.022	0.023
128 bits	0.022	0.023
448 bits	0.022	0.023

Table 19. Average Blowfish Encipher/Decipher Execution Times for Varying Key Sizes

Analysis of the CPU usage impact: This experiment was designed to identify the impact of *SecureFilter* operations on CPU usage. The *SecureFilter* was using the Blowfish algorithm with a 448-bit key size. The frequency of data packet production was manipulated to produce three different rates: 30, 60, and 120Hz. The CPU readings were taken by using the system performance tab of the Windows2000 Task Manager, and observing the minimum, maximum, and the predominant range during a two minute time frame. The results are provided in Table 20, and indicate that the impact of a single *SecureFilter* on CPU usage is negligible at less than four percent at 120Hz.

Impact of SecureFilter on CPU usage packet size: 156 bytes		
rate	**min/max**	**predominant**
30 Hz	0/3%	0-2%
60HZ	0/4%	0-3%
120HZ	1/4%	1-3%

Table 20. CPU Usage during *SecureFilter* operation

E. SEQUENCE FILTER DELAY STUDY

We also explored the effects of the sequencing operations of the filter. We did not expect the delay introduced by this filter to be significant.

1. Study Design

As identified in section C of this chapter, the outbound algorithm in the *sendPacket* method and inbound algorithm in the *packetReceived* method are encapsulated in a *for*-loop. This experiment was also performed on data array sizes of 156 and 116 bytes. Note that the data sizes in the charts of this section indicate the size of the data arrays that are passed through the transmit and receive algorithms; in the case of 156/168, 156 bytes are entering the transmit algorithm, while 168 bytes (four-byte sequence number plus eight-byte application ID plus 156 byte data array) enter the receive algorithm.

2. Results

Results of this experiment are shown in Table 21 below. The results indicate the following:

- Overall execution time of the transmit algorithm is 0.002 milliseconds, and is not dependent on the size of the data that is provided.
- Overall execution time of the receive algorithm is 0.004 milliseconds. The apparent difference in times due to data size is curious, yet not significant. There is nothing in the code that can account for this.
- The reception times are twice that of the transmittal times; this is due to accessing of the hash tables to verify the application ID and sequence number of the incoming packets.

- An average of 0.006 milliseconds is required for sequencing operations to be performed on a data packet from one host to another.
- The impact of this filter on delay is negligible.

SequenceFilter-induced delay (milliseconds)			
Data size (bytes)	Transmit	Receive	Total Time
116/128	0.002	0.004	0.006
156/168	0.002	0.004	0.006
Difference	0.000	0.001	0.001

Table 21. Average Execution Times for *SequenceFilter*'s transmit and receive Algorithms

F. INTEGRITY FILTER DELAY STUDY

We also investigated the effects of the sequencing operations of the filter. We did not expect to observe a significant delay.

1. Study Design

The outbound algorithm in the *sendPacket* method and inbound algorithm in the *packetReceived* method are encapsulated in a *for*-loop. This experiment was also performed on data array sizes of 156 and 116 bytes. Note that the data sizes in the charts of this section indicate the size of the data arrays that are passed through the transmit and receive algorithms; in the case of 156/180, 156 bytes enter the transmittal algorithm, while 180 bytes (four-byte length plus twenty byte message digest plus 156 byte data array) enter the receive algorithm.

2. Results

Results of this experiment are shown in Table 22. The results of the experiment indicate the following:

- Overall execution time of the transmit algorithm ranges from 0.012 to 0.018 milliseconds for an array of 116 and 156 bytes, respectively and appears to be dependent on the size of the data array provided to the *MessageDigest* object, as expected.

- Overall execution time of the receive algorithm ranges from 0.014 to 0.020 milliseconds for an array of 140 and 180 bytes, respectively. This also appears to be dependent on the data array size.
- The reception times are approximately 0.002 milliseconds slower that the transmit algorithm, probably due to the message digest comparison call.
- An average of 0.026 milliseconds is required for integrity operations to be performed on a 116-byte data packet from one host to another.
- An average of 0.038 milliseconds is required for integrity operations to be performed on a 156-byte data packet from one host to another.
- The impact of this filter on delay is negligible

IntegrityFilter-induced delay (milliseconds)			
Data size (bytes)	Transmit	Receive	Total Time
116/140	0.012	0.014	0.027
156/180	0.018	0.020	0.038
Difference	0.006	0.006	0.012

Table 22. Average Execution Times for *IntegrityFilter*'s Transmit/Receive Algorithms

G. ANALYSIS OF OVERALL DELAY IMPACT

Each of the filters imposes some delay on the overall transmission of an individual packet. An analysis of the total time delay that is induced on a packet by all the filters combined is provided in Table 23. As the total delay column indicates, the delay imposed on a packet of 116 bytes ranges from 0.082 to 0.220 milliseconds; and the delay for a packet of 156 bytes ranges from 0.111 to 0.295 milliseconds.

Cipher Algorithm: DES				
Data Size	SecureFilter	SequenceFilter	IntegrityFilter	Total delay
116	0.077	0.006	0.027	0.109
156	0.102	0.006	0.038	0.146
Cipher Algorithm: DESede				
Data Size	SecureFilter	SequenceFilter	IntegrityFilter	Total delay
116	0.188	0.006	0.027	0.220
156	0.250	0.006	0.038	0.295
Cipher Algorithm: Blowfish				
Data Size	SecureFilter	SequenceFilter	IntegrityFilter	Total delay
116	0.050	0.006	0.027	0.082
156	0.066	0.006	0.038	0.111

Table 23. Total Time Delay Induced by all Filters per Cipher algorithm

Since the filters were not tested under heavy packet transmission conditions, it is difficult to say if the delays identified in these studies will continue to hold for other workloads. Further tests need to be conducted to characterize these delays. As the data indicates, the impact of the all the filters combined on delay is less than 0.3 for DESede and less than 0.12 milliseconds for Blowfish. The use of these security measures appears to require very little overhead.

H. ANALYSIS OF OVERALL BANDWIDTH IMPACT

Each of the filters increases the size of the data packet that is transmitted across the network. These increases accumulate and may impact the required bandwidth for a specific RTEVE system. As identified in Chapter IV, and presented in Table 24, each of the filters impact the size of the data packet in differing amounts, based on the performed operations.

Filter	Increase to Data Array Size
SecureFilter	4 - 11 bytes
SequenceFilter	12 bytes
IntegrityFilter	21 bytes

Table 24. Filter Impact on Data Array Size

If all the filters are placed in series, the total increase to a data array would be thirty-seven to forty-four bytes. Depending on the application structure this could be quite an increase in bandwidth requirement (i.e., if the average packet were 100 bytes in length, it has now been increased by 44%) or minimal (i.e. if the average packet were 300 bytes in length, then the increase is only 15%). Either way, the main impact would be a function of the average number of packets transmitting at any point in time. If the average number of packets on the network is 10,000 per second, as is anticipated for a large scale VE with 1000 participants transmitting 10 packets per second, then the overhead alone for using the three filters would be increased by 440,000 bytes (3,520,000 bits); this is not an insignificant amount of bandwidth particularly if dealing with low-bandwidth communication lines. However, since most VE systems currently use some form of dead reckoning algorithm to reduce the amount of packet transmissions, the overhead caused by the increased packet size is most likely tolerable. Moreover, with communication lines reaching 10Gbps, the impact may be negligible in some environments.

Considering that most sensitive RTEVE applications requiring high levels of security will tend to have dedicated high-bandwidth communication lines, the impact in these environments would be minimal. A detailed analysis needs to be conducted in this area to better determine the impact in the general case.

I. SUMMARY

This chapter presented an analysis of the three types of filter objects that were designed as part of the NSMS. The analysis covered the impact that these filters had on the areas of delay and bandwidth. This analysis is basic in scope and should be expanded before firm conclusions are made as to their impact on QOS. However, it appears that the impact on delay is negligible, while the increase in required bandwidth is acceptable and CPU usage is within acceptable limits. The following chapter summarizes the conclusions of this work and discusses future work.

VI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The research reported in this thesis serves as the basis for exploring a wide spectrum of information assurance areas, as they relate to RTEVEs. We presented a taxonomy of RTEVE IA concerns, then explored some of the concerns in the context of a state-of-the-art RTEVE framework known as NPSNET-V.

1. NSMS

The design of the filters supports the selection of the level of security to be applied in each virtual world. The individual effects of the designed filters on delay and bandwidth were quantified. The results lead us to conclude that security features on data packet transmissions are technically feasible for large-scale VE development with negligible impact on delay and acceptable impact on bandwidth.

2. RTEVE Security

We discovered that the security of RTEVEs covers at least twenty-five different areas of information assurance. Some areas have been significantly addressed by current research, while others have received little attention. Nonetheless, each area must be addressed in some fashion whenever an RTEVE is developed, even if the decision is to not address it. The level of security that is desired for an RTEVE must be identified by the intended user of the system, and then the system developed to those requirements.

3. RTEVE Security System

A comprehensive RTEVE security system needs to have the ability to manage the RTEVE no matter how large it grows. For a system such as NPSNET-V, with a goal of infinite scalability, this is a formidable challenge. Secondly, it must contain a robust manner of detecting any intrusion whether known or novel, and successfully responding to neutralize the intrusion or minimize the impact, such as through intelligent management of the generation and distribution of encryption keys. However, there is still some likelihood that an attacker might find weaknesses in the security systems, or on the platforms in which the RTEVE resides. Therefore, we

conclude that the system should be updateable to allow for inclusion of new security policy and support mechanisms.

B. FUTURE WORK

The work accomplished in this thesis is merely a beginning to what can be a comprehensive distributed security management system for a VE. New technologies are constantly being researched and developed. The following are functionalities that can be incorporated into this work for trusted RTEVEs.

1. Perform a Comprehensive Statistical Analysis of the NSMS

As briefly discussed in the previous chapter, the studies performed in this work are not comprehensive in addressing the impact on QOS issues. A more thorough statistical analysis must be performed in order to identify the impact on QOS over ranges of system workloads. This study should involve multiple applications, with hundreds of clients where possible, that are producing increasingly large numbers of data packets.

2. Develop 'Distributedness' Capability of NSMS

The ability of the NSMS to maintain desired levels of QOS and security as an NPSNET-V application expands must be assured. A server-based structure is inefficient in a large-scale networked environment for many reasons, including the fact that it represents a single–point–of–failure. Also, key distribution to possibly thousands of participants is extremely inefficient, and would allow for a possibly compromised key to be active for an inordinate amount of time while a new key is distributed.

A good starting point for this area would be to use the idea of a tree structure [Yerry84] for security management distribution, in which the management of security issues and keys is shared amongst all nodes of the tree. Specific characteristics of this system would include:

- The tree concept would allow for fast key distribution amongst all nodes of the tree. The exponential increase in parallelism as the key traverses down the tree, with each node ensuring the distribution of the key to its siblings, allows for the operation of key distribution to be more efficient.

- The tree should be self-repairable; that is when a node is dropped from the tree, the tree structure below that node should be able to reconstitute itself back into the higher structure.
- The tree must be self-policing, that is, if any node is acting suspicious, such as constantly changing the keys of its children for no reason, its children should be able to identify it as a subverted node, and remove it and reconstitute the tree from that point on forward.
- Each node of the tree would be represented by a *SecurityManager* object within the NPSNET-V application that can act both as a server for the tree structure below it, and as the manager of filters for its own application.
- To facilitate key management, each node should be the key manager for all filters below it that are not known anywhere else in the tree. This way responsibility is maintained at the lowest level possible and does not unnecessarily burden higher level nodes in the tree. Each *SecurityManager* must be able to act as the server itself. Or, each module can act as a key distribution point for any filters below it that are not known anywhere else in the structure.

3. Intrusion Detection Capability

An addition to any comprehensive security policy is the inclusion of an intrusion detection system, either signature- or anomaly-based. The benefits of a good anomaly-based system are obviously great and are preferred to those of a signature. The IDS system concepts discussed in [Vigna98] and [Stillerman99] are an interesting place to proceed from. Unfortunately, anomaly-based IDSs are still in their infancy and, therefore, beginning with a signature-based IDS to provide known intrusion detection would be a lower risk approach.

4. Intrusion Response Capability

Along with an intrusion detection capability, the system should possess an effective and efficient response capability in order to effectively protect the system and minimize effects of an attack. Responses could take the form of denial of future connectivity to a malicious application, dynamic key changes in response to a discovered

compromise of the symmetric keys, or even the use of software decoys in order to learn more information about the attacker and the nature of the intrusion [Michael02].

5. Increase Functionality of the NSMS

The functionality of the NSMS can be expanded in many directions, such as the following:

- Apply module integrity and authentication through the use of Jar-signing, message digests, or checksums.
- Increase the capability of the filters by allowing the *SecureServer* to manipulate the message digest algorithms in the *IntegrityFilter*.
- Connect the NSMS to a network monitor that informs the *SecureServer* of the current level of latency within the system, for the purpose of dynamically adjusting the generation and distribution of keys for the entire network with the aim of minimizing the period of key mismatch.
- Incorporate an audit log that can be used for post-intrusion detection efforts. The concerns that must be managed here are the trade-offs between amount of events that are recorded, amount of memory space available to use as a record file, and the rate at which the files are scanned for malicious activity. Also, an adequate security measure needs to be provided for the log itself to prevent hackers from erasing their activities from the log.
- Increase the number of key algorithms to choose from, and the breadth of chaining modes and padding schemes. Other providers have encryption packages that can be included for use with the system.
- Have the sequence filter verify authenticity of new application IDs that appear on incoming packets with the *SecureServer*.
- Address the security weaknesses identified in section I of chapter IV.

APPENDIX

A. SAMPLE CONFIGURATION FILE

```
<?xml version="1.0"?>
<Configuration>
    <Header>
        <Meta name="description" content="DIS networking test (first client)."/>
        <Meta name="author" content="Andrzej Kapolka"/>
    </Header>
    <Body>
        <Include url="../include/base.xml"/>
        <Include url="../include/gui.xml"/>
        <Include url="../include/dis.xml"/>
        <Container name="client_a">
            <World name="org/npsnet/v/worlds/examples/EmptyWorld.xml"
                   modelName="modelCore">
                <Entity name="org/npsnet/v/entities/cameras/PilotableCamera.xml"
                        modelName="pilotableCamera"/>
                <Entity name="org/npsnet/v/entities/examples/Teapot.xml"
modelName="teapot_a">
                    <Transform translation="-5 0 -20"/>
                </Entity>
            </World>
            <Module class="org.npsnet.v.views.j3d.J3DViewCore">
                <Target name="modelCore"/>
                <Viewport title="NPSNET-V: DIS Networking Test (Client A)"
                          xPos="128" camera="modelCore/pilotableCamera"/>
            </Module>
            <Module class="org.npsnet.v.controllers.user.AWTControllerCore">
                <Target name="modelCore"/>
                <Module
class="org.npsnet.v.controllers.user.MouseContextMenuController"/>
                <Module class="org.npsnet.v.controllers.user.MouseTransformController"/>
            </Module>
            <Module class="org.npsnet.v.controllers.network.dis.DISControllerCore">
                <Target name="modelCore"/>
                <Module class="org.npsnet.v.channels.AggregatingChannel">
                    <Module class="org.npsnet.v.channels.MulticastChannel">
                        <Address value="225.93.23.93"/>
                    </Module>
                </Module>
            </Module>
        </Container>
    </Body>
```

This configurations file generates a world that first incorporates three base functionality architectures identified as ‘Base.xml”, “gui.xml”, and “dis.xml.” These are attached beneath the Kernel of the system. Below the Kernel, another container called “client a” is created. This container holds four modules below it; those are: “EmptyWorld”, which contains a “pilotableCamera” entity and a “teapot” entity; “j3DViewCore”, which is associated with the pilotableCamera; “AWTControllerCore”, which contains a “MouseContextMenuController” and a “MouseTransformController” that are used for entity manipulation; and the “DISControllerCore”, which contains an “AggregatingChannel”, which contains a “MulticastChannel’ module.

This file will generate a world with a teapot that can be controlled with the mouse, and communicates using DIS protocols through a multicast channel.

THIS PAGE INTENTIONALLY LEFT BLANK

B. NPSNET-V SOURCE CODE

It should be noted that the NSMS system developed in this thesis is not a standalone work. Components of the system are spread throughout the module areas of NPSNET-V and can function only as modules of NPSNET-V. The complete NPSNET-V system, including source code, documentation, and points of contact are located in a CVS repository at following webpage:

http://sourceforge.net/projects/npsnetv/

NPSNET-V is constantly in progress. Contributors and developers are welcome to join in the development efforts associated with this RTEVE project. Contact one of the Project Administrators for information on the current status of the system and how to become a contributor.

The below table identifies the locations of the NSMS source files discussed in this work within the NPSNET-V file structure.

Source File	Location
SecureServer	servers
SecureServerGUI	servers
SecureServerConnection	servers
KeyMaker	servers
SecureFilter	channels
IntegrityFilter	channels
SequenceFilter	channels
StandardSecurityManager	system
SecurityManager	services\system
SecurityManagerSubscriber	services\system
SecretKeyPack	services\system
All locations are relative to the directory: npsnetv\source\org\npsnet\v\	

THIS PAGE INTENTIONALLY LEFT BLANK

C. SAMPLE NSMS XML CONFIGURATION FILE

```
<?xml version="1.0"?>
<Configuration>
   <Header>
      <Meta name="description" content="NSMS secure test B"/>
      <Meta name="author" content="Andrzej Kapolka"/>
   </Header>
   <Body>
      <Include url="../include/base.xml"/>
      <Include url="../include/gui.xml"/>
      <Include url="../include/dis.xml"/>
      <Module class="org.npsnet.v.system.StandardSecurityManager"/>
      <Container name="client_b">
         <World name="org/npsnet/v/worlds/examples/EmptyWorld.xml"
               modelName="modelCore">
            <Entity name="org/npsnet/v/entities/cameras/PilotableCamera.xml"
                  modelName="pilotableCamera"/>
            <Entity name="org/npsnet/v/entities/munitions/antiship/SphericalMine.xml" modelName="teapot_b">
               <Transform translation="2 0 -20"/>
            </Entity>
            <Entity name="org/npsnet/v/entities/environment/StandardExplosionManager.xml"/>
         </World>
         <Module class="org.npsnet.v.views.j3d.J3DViewCore">
            <Target name="modelCore"/>
            <Viewport title="NPSNET-V: DIS Networking Test (Client B)"
                  xPos="656" camera="modelCore/pilotableCamera"/>
         </Module>
         <Module class="org.npsnet.v.controllers.user.AWTControllerCore">
            <Target name="modelCore"/>
            <Module class="org.npsnet.v.controllers.user.MouseContextMenuController"/>
            <Module class="org.npsnet.v.controllers.user.MouseTransformController"/>
         </Module>
         <Module class="org.npsnet.v.controllers.network.dis.DISControllerCore">
            <Target name="modelCore"/>
            <Module class="org.npsnet.v.channels.SequenceFilter">
                                    <Set name="ID" value = "seq92"/>
               <Module class="org.npsnet.v.channels.IntegrityFilter">
                                       <Set name="ID" value = "int92"/>
                  <Module class="org.npsnet.v.channels.SecureFilter">
                                          <Set name="ID" value = "sec92"/>
                     <Module class="org.npsnet.v.channels.MulticastChannel">
                        <Address value="225.93.23.92"/>
                     </Module>
                  </Module>
               </Module>
            </Module>
            <Module class="org.npsnet.v.channels.SecureFilter">
                                    <Set name="ID" value = "sec96"/>
               <Module class="org.npsnet.v.channels.MulticastChannel">
                  <Address value="225.93.23.96"/>
               </Module>
            </Module>
         </Module>
      </Container>
   </Body>
```

THIS PAGE INTENTIONALLY LEFT BLANK

D. SEQUENCE FILTER CODE

```
package org.npsnet.v.channels;

import java.io.*;
import java.util.*;
import java.security.*;
import java.security.spec.*;
import java.util.Timer;
import java.util.TimerTask;

import javax.crypto.*;
import javax.crypto.spec.*;

import org.npsnet.v.kernel.Module;
import org.npsnet.v.kernel.ModuleContainer;
import org.npsnet.v.kernel.ModuleContainerEvent;
import org.npsnet.v.kernel.PropertyBearerListener;
import org.npsnet.v.kernel.PropertyBearerRegistrationEvent;
import org.npsnet.v.kernel.PropertyBearerDeregistrationEvent;

import org.npsnet.v.properties.channel.Channel;
import org.npsnet.v.properties.channel.DataPacket;
import org.npsnet.v.properties.channel.ReceivedPacketListener;

import org.npsnet.v.services.system.SecurityManagerSubscriber;
import org.npsnet.v.services.system.SecretKeyPack;
import org.npsnet.v.services.system.SecurityManager;

/**
 * A filter channel that performs Sequencing functions on the passed packets.
 * The outbound packets have the 8-byte application ID and a 4-byte integer
 * sequence number appended to the data before transmittal.
 *  The sequence number is initialized to a random number in order to prevent
 *  guessing by an attacker.
 *
 * incoming packets are verified for proper sequencing.  If the packet is the
 *   first one seen from a previously unknown application, the filter wiil
 *   register the application an dbegin tracking the sequence numbers.
 *   the sequence number of any new packet must be at most 20 units greater than
 *   the previously seen packet for that applicationID.  RollOver is followed,
 *   that is when an application reaches the maximum integer value, it will
 *   'rollover' and begin at the lowest integer value.
 * This class can only function if a SecurityManager object is included in
 *  the application structure.
 *  Does not register with the securityManager until the setID method is called
 *  by the XML confiduration file
 *
 * Any out of sequence and repeat packets will be dropped, thus addressing the
 * replay attack.
 *
 * the 'filter' functionality is based on the aggregate filter authored
 * by Andrzej Kapolka
 *
 * @author Ernesto Salles
 */

public class SequenceFilter extends ModuleContainer
                            implements Channel, ReceivedPacketListener,
                                       PropertyBearerListener,
                                       SecurityManagerSubscriber
{
    /**
    * boolean used to signal the filter to transmit
    */
    boolean okToTransmit;

    /**
     * boolean used to signal the filter to receive
     */
    boolean okToReceive;
```

```
/**
 * the sequence number for outbound packets from this filter
 */
int seqNum;

/**
 * The application ID
 */
long applicationID;

/**
 * The filter's ID
 */
String filterID;

/**
 * The list of channel listeners.
 */
private Vector receivedPacketListeners;

/**
 * The SecurityManager
 */
private SecurityManager securityManager;

/**
 * the outputstream used to generate byte arrays
 */
ByteArrayOutputStream baos;

/**
 * Used to input data into the baos
 */
DataOutputStream dos;

/**
 * a table containing applications that are communicatig on this comms path,
 *  and their sequence numbers for replay attack prevention
 */
Hashtable appSequenceTable;

/**
 * Constructor.  Initialize the required objects and turns the filter off
 */
public SequenceFilter()
{
    receivedPacketListeners = new Vector();
    this.endPacketTransmission();
    this.endPacketReception();
    applicationID = 0;
    Random rnd = new Random();
    seqNum = rnd.nextInt();
    this.println("initial sequence number is: " + seqNum);
    appSequenceTable = new Hashtable();
    try{
      baos = new ByteArrayOutputStream();
      dos = new DataOutputStream(baos);
    }
    catch(Exception e){}
}

/**
 * Initializes this module.
 */
public void init(){
  // Register with self as registration listener
  addPropertyBearerListener(Channel.class,this);
  super.init();
}
```

```
/**
 * Takes the given packet, and adds the application ID and a sequence number
 * to the data then sends the new packet on its way
 *
 * @param packet the packet to be sent
 * @exception IOException if an error occurs
 */
public synchronized void sendPacket(DataPacket packet) throws IOException
{

  // if the filter is ok to transmit
  if (okToTransmit){
    byte[] dataArray = new byte[1];
    try{

      //gets the data from the packet, and appends the applicationID and sequence
      //   number to the beginning
      byte[] packetData = packet.getData();
      baos.reset();
      dos.writeLong(applicationID);
      dos.writeInt(seqNum);
      dos.write(packetData,0,packetData.length);
      dataArray = baos.toByteArray();

      // if the sequnce number is the Maximum Integer Value, then assign it the
      //   Minimum integer Value, else increment it.  This 'roll-over' ensures
      //  sequence number continuity when the maximum value has been reached
      if(seqNum == Integer.MAX_VALUE){
        seqNum = Integer.MIN_VALUE;
        this.println("Sequence Number rollover");
      }
      else {seqNum++;}
    }
    catch(Exception e){}

    // create a new packet with the enciphered data
    DataPacket dp = new DataPacket(dataArray);
    dp.setLength(dataArray.length);
    dp.setOffset(0);

    // send the enciphered data packet out to the listening channels
    Enumeration enum = getPropertyBearers(Channel.class);
    while(enum.hasMoreElements())
    {
      try
      {
        ((Channel)enum.nextElement()).sendPacket(dp);
      }
      catch(IOException ioe)
      {
        System.out.println(ioe);
      }
    }
  }
}

/**
 * Called when a packet is received.
 *
 * @param c the channel on which the packet was received
 * @param packet the received packet
 */
public void packetReceived(Channel c, DataPacket packet)
{
  try{
    if (okToReceive){

      // extract the data from the packet into the correctly sized array
      byte[] dataArray = new byte[packet.getLength()];
      System.arraycopy(packet.getData(), 0,
                       dataArray, 0, packet.getLength());
```

```
      //extract the applicationID, sequence number, and clear data from the
      //   deciphered data array
      ByteArrayInputStream bais = new ByteArrayInputStream(dataArray);
      DataInputStream dis = new DataInputStream(bais);
      long appID = dis.readLong();
      int sequenceNum = dis.readInt();
      byte[] newData = new byte[dis.available()];
      dis.read(newData);

      // check the sequence number & ID of the packet
      boolean packetOK = this.checkIDandSequence(appID,sequenceNum);

      // if there actually is data, and the packet is ok to send, then
      //   package it into a new packet and send it on
      if ((newData != null) &&(packetOK) ){
        DataPacket dp = new DataPacket(newData);
        dp.setLength(newData.length);
        dp.setOffset(0);

        Enumeration enum = receivedPacketListeners.elements();
        while(enum.hasMoreElements())
        {
          ((ReceivedPacketListener)enum.nextElement()).packetReceived(this,dp);
        }
      }
    }
  }
  catch(Exception e){
    System.out.println(e);
  }
}


/**
 * checks the sequence number of the given applicationID to prevent replay
 *
 * @param appID   the applicationID
 * @param seqNum  the sequence number
 *
 * @return boolean true if the sequence # is good,
 *                 false if the sequence number is bad
 */
 private boolean checkIDandSequence(long appID, int seqNum){
  Long appIDNum = new Long(appID);
  Integer seqNumber = new Integer(seqNum);

  // if the packet is this application's own packet, reject it
  if (applicationID == appID) {
    return false;
  }

  // if the packet's application ID is already known, then check the
  //   sequence number
  if(appSequenceTable.containsKey(appIDNum)){
    Integer lastSeq = (Integer) appSequenceTable.get(appIDNum);
    int lastSeqNum = lastSeq.intValue();

    // if the last packet's sequence number is less than this packet's
    //    sequence number by 20, then the packet is good; replace the sequence
    //    number in the table with the new one
    int difference = (seqNum - lastSeqNum);

    // this covers all cases if the last sequence number is below
    //    (Max Integer value - 20), and if both sequence numbers
    //    are greater than 0.
    if((difference > 0) && (difference < 20)){
      Integer temp = (Integer)appSequenceTable.put(appIDNum,seqNumber);
      return true;
    }
```

```
    // this covers the times when the old sequence number is greater than
    //    (Max Integer value - 20), and the new number is less than zero
    //     ie. the rollover area
    else if (((Integer.MAX_VALUE - lastSeqNum) < 20) && (seqNum < 0)){
      int buffer = Integer.MAX_VALUE - lastSeqNum;
      if(seqNum < (Integer.MIN_VALUE + (20-buffer))){
        Integer temp = (Integer)appSequenceTable.put(appIDNum,seqNumber);
        this.println(" sequence rollover for app " + appID);
        return true;
      }
    }
    // else the packet is out of sequence and is rejected
    else {
      this.println("out-of-order sequence number; ");
      System.out.println("old: " + lastSeq.intValue() + " new: " + seqNum);
      return false;
    }
  }

  // if this is the first time seeing this application ID, then place the
  //   data into the table
  else{
    this.println("new ApplicationID discovered; adding to table");
    appSequenceTable.put(appIDNum,seqNumber);
  }
  return true;
}


/**
 * Adds a listener to the list of objects interested
 * in incoming packets.
 *
 * @param rcl the listener object to add
 */
public void addReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.add(rcl);
}

/**
 * Removes a listener object from the list of objects
 * interested in incoming packets.
 *
 * @param rcl the listener object to remove
 */
public void removeReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.remove(rcl);
}

/**
 * Invoked when a property bearer is registered.
 *
 * @param pbre the event object
 */
public void propertyBearerRegistered(PropertyBearerRegistrationEvent pbre)
{
    ((Channel)pbre.getRegisteredPropertyBearer()).addReceivedPacketListener(this);
}

/**
 * Invoked when a property bearer is deregistered.
 *
 * @param pbde the event object
 */
public void propertyBearerDeregistered(PropertyBearerDeregistrationEvent pbde)
{
    ((Channel)pbde.getDeregisteredPropertyBearer()).
                                          removeReceivedPacketListener(this);
}
```

```
/**
 * Begins the transmission of packets
 */
public void beginPacketTransmission(){
  okToTransmit = true;
  this.println("begin transmit **");
}

/**
 * Stops the transmission of packets
 */
public void endPacketTransmission(){
  okToTransmit = false;
  this.println("end transmit **");
}
/**
 * Begins the reception of packets
 */
public void beginPacketReception(){
  okToReceive = true;
  this.println("begin receive **");
}

/**
 * Stops the reception of packets
 */
public void endPacketReception(){
  okToReceive = false;
  this.println("end receive **");
}

/**
 * Sets the applications' ID
 *
 * @param id  the application's id
 */
public void setApplicationID(long id){
  applicationID = id;
  this.println("Application ID set: " + applicationID);
}

/**
 * sets the Filter's ID, and register the filter with the security manager
 *
 * @param  id  the filter's id
 */
public void setID(String id){
  filterID = id;
  this.println("filter ID set to: " + filterID);
  this.println("registering with SM");
  securityManager = (SecurityManager) getServiceProvider(SecurityManager.class);
  securityManager.addSecureSubscriber(this);
}

/**
 * gets the filter's ID
 *
 * @return String the filter's id
 */
 public String getID(){
  return filterID;
 }
```

```
    /**
     * returns the filter's type as identified in the
     *  SecurityManagerSubscriber Interface
     *
     * @return int  the filter's type
     */
    public int getFilterType(){
      return this.SEQUENCE_FILTER_TYPE;
    }

    /**
     * An empty method. must be implemented due to interface
     */
    public SecretKeyPack getCurrentKeyPack(){
      return null;
    }

    /**
     * An empty method. must be implemented due to interface
     */
    public Vector getAllKeyPacks() {
      return null;
    }

    /**
     * An empty method. must be implemented due to interface
     */
    public synchronized void addKeyPack(SecretKeyPack keyPack){
    }

    /**
     * Synonymous with 'System.out.println', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    private void println(String aLine){

      // call the 'print' method, passing the provided String object
      //   concatenated with a carriage return
      this.print(aLine.concat("\n"));
    }

    /**
     * Synonymous with 'System.out.print', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    private void print(String aLine){
      System.out.print("  Sequence Filter (" + filterID + "): " + aLine);
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

E. INTEGRITY FILTER CODE

```
package org.npsnet.v.channels;

import java.io.*;
import java.util.*;
import java.security.*;

import org.npsnet.v.kernel.Module;
import org.npsnet.v.kernel.ModuleContainer;
import org.npsnet.v.kernel.ModuleContainerEvent;
import org.npsnet.v.kernel.PropertyBearerListener;
import org.npsnet.v.kernel.PropertyBearerRegistrationEvent;
import org.npsnet.v.kernel.PropertyBearerDeregistrationEvent;

import org.npsnet.v.properties.channel.Channel;
import org.npsnet.v.properties.channel.DataPacket;
import org.npsnet.v.properties.channel.ReceivedPacketListener;

import org.npsnet.v.services.system.SecurityManagerSubscriber;
import org.npsnet.v.services.system.SecretKeyPack;
import org.npsnet.v.services.system.SecurityManager;


/**
 * A filter channel that performs integrity operations on data packets.  It uses
 * the MessageDigest object to generate and verify message digests.  This
 * implementation uses the default algorithm of SHA, which produces a 20-byte
 * message digest.  The algorithm also places one-byte in the beginning of the
 * data array that indicates the size of the digest.
 *
 * This class can only function if a SecurityManager object is included in
 *  the application structure.
 * Does not register with the securityManager until the setID method is called
 * by the XML configuration file
 *
 * the 'filter' functionality is based on the aggregate filter authored
 * by Andrzej Kapolka
 * @author Ernesto Salles
 */

public class IntegrityFilter extends ModuleContainer
                             implements Channel, ReceivedPacketListener,
                                        PropertyBearerListener,
                                        SecurityManagerSubscriber{

    /**
    * boolean used to signal the filter to transmit
    */
    boolean okToTransmit;

    /**
     * boolean used to signal the filter to receive
     */
    boolean okToReceive;

    /**
     * The application ID
     */
    long applicationID;

    /**
     * The filter's ID
     */
    String filterID;

    /**
     * The list of channel listeners.
     */
    private Vector receivedPacketListeners;
```

```
/**
 * The SecurityManager
 */
private SecurityManager securityManager;

/**
 * the stream used to create byte arrays
 */
ByteArrayOutputStream baos;

/**
 * used to input data into the baos
 */
DataOutputStream dos;

/**
 * message digest generator used for transmission
 */
MessageDigest transmitMessageDigester;

/**
 * message digest generator used for verifying incomming data
 */
MessageDigest messageDigestChecker;

/**
 * Constructor.  initializes the required objects and turns off the filter
 */
public IntegrityFilter()
{
  try{
    transmitMessageDigester = MessageDigest.getInstance("SHA");
    messageDigestChecker = MessageDigest.getInstance("SHA");
    receivedPacketListeners = new Vector();
    this.endPacketTransmission();
    this.endPacketReception();
    applicationID = 0;
    baos = new ByteArrayOutputStream();
    dos = new DataOutputStream(baos);
  }
  catch(Exception e){}
}

/**
 * Initializes this module.
 */
public void init()
{
  // Register with self as registration listener
  addPropertyBearerListener(Channel.class,this);
  super.init();
}

/**
 * Sends a packet over this channel.
 *
 * @param packet the packet to be sent
 * @exception IOException if an error occurs
 */
public synchronized void sendPacket(DataPacket packet) throws IOException
{
  // if it is ok to transmit then create the digest and new packet and send it
  if (okToTransmit){
    byte[] dataArray = new byte[1];
    try{

      //gets the data from the packet, then feeds it to the message digest
      //   generator and retrieve the digest
      byte[] packetData = packet.getData();
      transmitMessageDigester.update(packetData);
      byte[] digest = transmitMessageDigester.digest();
```

```
      // reset the byteArrayOutputStream and generate the new byteArray,
      //  first the digest length, then the digest, and finally the data array
      //   Then get the new data array
      baos.reset();
      dos.writeByte(digest.length);
      dos.write(digest,0,digest.length);
      dos.write(packetData,0,packetData.length);
      dataArray = baos.toByteArray();
    }
    catch(Exception e){}

    // create a new packet with the data
    DataPacket dp = new DataPacket(dataArray);
    dp.setLength(dataArray.length);
    dp.setOffset(0);

    // send the data packet out to the listening channels
    Enumeration enum = getPropertyBearers(Channel.class);

    while(enum.hasMoreElements())
    {
      try
      {
        ((Channel)enum.nextElement()).sendPacket(dp);
      }
      catch(IOException ioe)
      {
        System.out.println(ioe);
      }
    }
  }
}

/**
 * Called when a packet is received.
 *
 * @param c the channel on which the packet was received
 * @param packet the received packet
 */
public void packetReceived(Channel c, DataPacket packet)
{
  try{
    if (okToReceive){

      // extract the data from the packet into the correctly sized array
      byte[] dataArray = new byte[packet.getLength()];
      System.arraycopy(packet.getData(), 0,
                       dataArray, 0, packet.getLength());

      //extract the digest length, digest, and data from the packet array
      ByteArrayInputStream bais = new ByteArrayInputStream(dataArray);
      DataInputStream dis = new DataInputStream(bais);
      int digestLength = new Byte(dis.readByte()).intValue();
      byte[] digest = new byte[digestLength];
      dis.read(digest);
      byte[] newData = new byte[dis.available()];
      dis.read(newData);

      // check the sequence number & ID of the packet
      boolean packetOK = this.verifyDigest(digest,newData);

      // if there actually is data, and the packet is ok to send, then
      //   package it into a new packet
      DataPacket dp = new DataPacket(newData);
      dp.setLength(newData.length);
      dp.setOffset(0);
```

```
      // if the packet is OK to keep sending, then send it
      if (packetOK){
        Enumeration enum = receivedPacketListeners.elements();
        while(enum.hasMoreElements())
        {
          ((ReceivedPacketListener)enum.nextElement()).packetReceived(this,dp);
        }
      }
      else {this.println("Integrity check failed");}
    }
  }
  catch(Exception e){
    e.printStackTrace();
  }
}

/**
 * Generates a digest of the provided data array, and compares it to
 *  the provided digest.  Returns true is the match, false if not
 *
 * @param digest    the digest to compare the new one to
 * @param data      the data to verify
 * @return boolean  the verdict
 */
private boolean verifyDigest(byte[] digest, byte[] data){
  messageDigestChecker.update(data);
  if (MessageDigest.isEqual(messageDigestChecker.digest(),digest)){
    return true;
  }
  return false;
 }

/**
 * Adds a listener to the list of objects interested
 * in incoming packets.
 *
 * @param rcl the listener object to add
 */
public void addReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.add(rcl);
}

/**
 * Removes a listener object from the list of objects
 * interested in incoming packets.
 *
 * @param rcl the listener object to remove
 */
public void removeReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.remove(rcl);
}

/**
 * Invoked when a property bearer is registered.
 *
 * @param pbre the event object
 */
public void propertyBearerRegistered(PropertyBearerRegistrationEvent pbre)
{
    ((Channel)pbre.getRegisteredPropertyBearer()).addReceivedPacketListener(this);
}
```

```
/**
 * Invoked when a property bearer is deregistered.
 *
 * @param pbde the event object
 */
public void propertyBearerDeregistered(PropertyBearerDeregistrationEvent pbde)
{
    ((Channel)pbde.getDeregisteredPropertyBearer()).
                                            removeReceivedPacketListener(this);
}

/**
 * Begins the transmission of packets
 */
public void beginPacketTransmission(){
  okToTransmit = true;
  this.println("begin transmit **");
}

/**
 * Stops the transmission of packets
 */
public void endPacketTransmission(){
  okToTransmit = false;
  this.println("end transmit **");
}

/**
 * Begins the reception of packets
 */
public void beginPacketReception(){
  okToReceive = true;
  this.println("begin receive **");
}

/**
 * Stops the reception of packets
 */
public void endPacketReception(){
  okToReceive = false;
  this.println("end receive **");
}

/**
 * Sets the applications' ID
 *
 * @param id  the application's ID
 */
public void setApplicationID(long id){
  applicationID = id;
  this.println("Application ID set: " + applicationID);
}

/**
 * sets the Filter's ID, and registers with the security manager
 *
 * @param this filter's ID
 */
public void setID(String id){
  filterID = id;
  this.println("filter ID set to: " + filterID);
  this.println("registering with SM");
  securityManager = (SecurityManager) getServiceProvider(SecurityManager.class);
  securityManager.addSecureSubscriber(this);
}
```

```
    /**
     * returns the filter's ID
     *
     * @return String   the filter's id
     */
    public String getID(){
      return filterID;
    }

    /**
     * returns the filter's type as identified in the
     *  SecurityManagerSubscriber Interface
     *
     * @return int   the filter's type
     */
    public int getFilterType(){
      return this.INTEGRITY_FILTER_TYPE;
    }

    /**
     * An empty method.  must be implemented due to interface
     */
    public SecretKeyPack getCurrentKeyPack(){
      return null;
    }

    /**
     * An empty method. must be implemented due to interface
     */
    public Vector getAllKeyPacks() {
      return null;
    }

    /**
     * an empty method.  must be implemented due to interface
     */
    public synchronized void addKeyPack(SecretKeyPack keyPack){
    }

    /**
     * Synonymous with 'System.out.println', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    public void println(String aLine){

      // call the 'print' method, passing the provided String object
      //   concatenated with a carriage return
      this.print(aLine.concat("\n"));
    }

    /**
     * Synonymous with 'System.out.print', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    public void print(String aLine){
      System.out.print("  Integrity Filter (" + filterID + "): " + aLine);
    }

}
```

F. SECURE FILTER CODE

```
package org.npsnet.v.channels;

import java.io.*;
import java.util.*;
import java.security.*;
import java.security.spec.*;
import java.util.Timer;
import java.util.TimerTask;

import javax.crypto.*;
import javax.crypto.spec.*;

import org.npsnet.v.kernel.Module;
import org.npsnet.v.kernel.ModuleContainer;
import org.npsnet.v.kernel.ModuleContainerEvent;
import org.npsnet.v.kernel.PropertyBearerListener;
import org.npsnet.v.kernel.PropertyBearerRegistrationEvent;
import org.npsnet.v.kernel.PropertyBearerDeregistrationEvent;

import org.npsnet.v.properties.channel.Channel;
import org.npsnet.v.properties.channel.DataPacket;
import org.npsnet.v.properties.channel.ReceivedPacketListener;

import org.npsnet.v.services.system.SecurityManagerSubscriber;
import org.npsnet.v.services.system.SecretKeyPack;
import org.npsnet.v.services.system.SecurityManager;
import org.npsnet.v.services.time.TimeProvider;

/**
 * A filter channel that performs encryption on the provided dataPackets.
 *
 * Cipher keys are provided to it through the use of SecretKeyPacks received from
 * the SecurityManager object with which it is associated.  This class can only
 * functions if a SecurityManager object is included in the application
 * structure.
 * Does not register with the securityManager until the setID method is called
 * by the XML confiduration file
 *
 * For outbound packets, it produces data arrays that are 4 to 11 bytes longer
 * than the original data:  4 bytes are added for the key ID, and 0-7 bytes are
 * added due to the padding scheme of the cipher.
 *
 * Key algorithms which are known to function within this class are DES, DESede,
 *  and Blowfish.  Only CBC or chaining mode has been verified fnctional.
 *
 * the 'filter' functionality is based on the aggregate filter authored
 * by Andrzej Kapolka
 *
 * @author Ernesto Salles
 */

public class SecureFilter extends ModuleContainer
                                implements Channel, ReceivedPacketListener,
                                           PropertyBearerListener,
                                           SecurityManagerSubscriber{

    /**
    * boolean used to signal the filter to transmit
    */
    boolean okToTransmit;

    /**
     * boolean used to signal the filter to receive
     */
    boolean okToReceive;
```

```
/**
 * The application ID
 */
long applicationID;

/**
 * The filter's ID
 */
String filterID;

/**
 * The list of channel listeners.
 */
private Vector receivedPacketListeners;

/**
 * Timer used in scheduling key change evolutions
 */
private Timer keyChangeTimer;

/**
 * The current KeyPack
 */
SecretKeyPack currentKeyPack = null;

/**
 * The KeyPacks awaiting to begin use,by order of start time
 */
Vector nextKeyPacks;

/**
 * The current encryption key
 */
private SecretKey currentKey;

/**
 * The timer object used for key changes
 */
private TimeProvider timeProvider;

 /**
  * The SecurityManager
  */
 private SecurityManager securityManager;

/**
 * The outbound encipher engine
 */
Cipher encipherer;

/**
 * The inbound decipher engine
 */

Cipher decipherer;

/**
 * Constructor.  initializes the objects of the filter and turns the filter off.
 */
public SecureFilter()
{
    receivedPacketListeners = new Vector();
    nextKeyPacks = new Vector();
    keyChangeTimer = new Timer();
    this.endPacketTransmission();
    this.endPacketReception();
    applicationID = 0;
    currentKeyPack = null;
}
```

```
/**
 * Initializes this module.
 */
public void init()
{
    // Register with self as registration listener

    addPropertyBearerListener(Channel.class,this);
    timeProvider = (TimeProvider) getServiceProvider(TimeProvider.class);
  super.init();
}

/**
 * encrypt a given packet and send it out to the next channel object.
 *
 * @param packet the packet to be sent
 * @exception IOException if an error occurs
 */
public synchronized void sendPacket(DataPacket packet) throws IOException
{

  // if the filter is ok to transmit, then encipher the data and transmit
  if (okToTransmit){

    // encrypt the packet's datapassed data array
    byte[] cipherArray  = encipherData(packet.getData());

    // create a new packet with the enciphered data
    DataPacket dp = new DataPacket(cipherArray);
    dp.setLength(cipherArray.length);
    dp.setOffset(0);



    // send the new packet out to the listening channels
    Enumeration enum = getPropertyBearers(Channel.class);
    while(enum.hasMoreElements())
    {
      try
      {
        ((Channel)enum.nextElement()).sendPacket(dp);
      }
      catch(IOException ioe)
      {
        System.out.println(ioe);
      }
    }
  }
}

/**
 * Called when a packet is received.
 *
 * @param c the channel on which the packet was received
 * @param packet the received packet
 */
public synchronized void packetReceived(Channel c, DataPacket packet)
{
  try{
    if (okToReceive){

      // extract the data from the packet into the correctly sized array
      byte[] cypherArray = new byte[packet.getLength()];
      System.arraycopy(packet.getData(), 0,
                       cypherArray, 0, packet.getLength());

      // decipher the data array
      byte[] clearData = decipherData(cypherArray);

      // if there actually is data, then
      //   package it into a new packet and send it up
```

```
      if (clearData != null){
        DataPacket dp = new DataPacket(clearData);
        dp.setLength(clearData.length);
        dp.setOffset(0);

        Enumeration enum = receivedPacketListeners.elements();
        while(enum.hasMoreElements())
        {
          ((ReceivedPacketListener)enum.nextElement()).packetReceived(this,dp);
        }
      }
    }
  }
  catch(Exception e){
    System.out.println(e);
  }
}

/**
 * Adds a listener to the list of objects interested
 * in incoming packets.
 *
 * @param rcl the listener object to add
 */
public void addReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.add(rcl);
}

/**
 * Removes a listener object from the list of objects
 * interested in incoming packets.
 *
 * @param rcl the listener object to remove
 */
public void removeReceivedPacketListener(ReceivedPacketListener rcl)
{
    receivedPacketListeners.remove(rcl);
}

/**
 * Invoked when a property bearer is registered.
 *
 * @param pbre the event object
 */
public void propertyBearerRegistered(PropertyBearerRegistrationEvent pbre)
{
    ((Channel)pbre.getRegisteredPropertyBearer()).addReceivedPacketListener(this);
}

/**
 * Invoked when a property bearer is deregistered.
 *
 * @param pbde the event object
 */
public void propertyBearerDeregistered(PropertyBearerDeregistrationEvent pbde)
{
    ((Channel)pbde.getDeregisteredPropertyBearer()).
                                  removeReceivedPacketListener(this);
}

/**
 * receives a SecretKeypack and either immediately installs it as the
 * current active key pack, or places it into the vector of future keypacks
 * and set's the timer for its active period
 *
 * @param  keyPack  the SecretKeyPack to add to the the vector
 */
public synchronized void addKeyPack(SecretKeyPack keyPack){
  if (keyPack != null){
    long keyLifeDuration = keyPack.getEndTime() - timeProvider.getCurrentTime();
```

```
    // if the packet has not expired, then schedule the key change, else reject it.
    if(keyLifeDuration>0){
      long keyStartDelay = keyPack.getBeginTime() - timeProvider.getCurrentTime();

      // if there is a delay before the beginning of this key, add the
      //  keyPack to the Vector and generate a TimerTask for the key to be
      //  changed to this KeyPack.  If the Vector is empty, then first
      //  generate a new Timer object in order to schedule the key change.
      if (keyStartDelay > 0) {
        if(nextKeyPacks.isEmpty()){
          keyChangeTimer = new Timer();
        }
        addPackToVector(keyPack);
        TimerTask keyChange = new TimerTask(){
          public void run(){changeKeys();}
        };
        keyChangeTimer.schedule(keyChange,keyStartDelay);
        this.println("a key pack has been received;  key change in "
                     + keyStartDelay + " milliseconds");
      }

      // if it's an immediate key change, then drop all awaiting packets,
      //    cancel the tasks in the Timer, and load the new pack into the
      //    nextPack Vector.  Then change the key.
      else {
        nextKeyPacks.clear();
        keyChangeTimer.cancel();
        nextKeyPacks.add(keyPack);
        this.println("an immediate key pack has been received;
                      immediate key change");
        this.println("****Vector has bee zeroed: " + nextKeyPacks.size());

        changeKeys();
      }
    }
    else {this.println("a Key pack has been rejected, time period has expired");}
  }
  else {this.println("Null keyPack received -> nothing done");}
}

/**
 * adds a new KeyPack to the Vector of keyPacks awaiting activation.  The pack
 *   is placed into the Vector in sequential order by time if activation
 *
 * @param newKeypack  the SecretKeyPack to add to the Vector
 */
private void addPackToVector(SecretKeyPack newKeyPack){
  SecretKeyPack skp1;
  int i;

  // go through the Vector and insert the ne KeyPack into its sequential
  //    location based on beginTime
  for(i = 0; i<nextKeyPacks.size(); i++){
    skp1 = (SecretKeyPack) nextKeyPacks.elementAt(i);
    if (newKeyPack.getBeginTime() < skp1.getBeginTime()){
      nextKeyPacks.add(i,newKeyPack);
      break;
    }
  }

  // if the loop has completed and the index is equal to the Vector size,
  //   then the keyPack wasn't added into the Vector, place it at the end
  if (i == nextKeyPacks.size()){
    nextKeyPacks.add(newKeyPack);
  }
}
```

```
/**
 * changes the keys.  First turns off the filter, then it removes the next
 * KeyPack off the Vector and assigns it as the current KeyPack.  It then
 * restarts the filter
 */
public void changeKeys(){
  endPacketTransmission();
  endPacketReception();
  if (currentKeyPack == null){currentKeyPack =
                              (SecretKeyPack)nextKeyPacks.elementAt(0);};
  synchronized (currentKeyPack){
    currentKeyPack = (SecretKeyPack)nextKeyPacks.remove(0);
    this.println("key removed from vector " + nextKeyPacks.size());
    this.println("key change, new key ID:        "
                 + new String(currentKeyPack.getKeyID()));
    this.println("key Cipher parameters are:     "
                 + currentKeyPack.getCipherParameters());

    try{
      encipherer = Cipher.getInstance(currentKeyPack.getCipherParameters());
      encipherer.init(Cipher.ENCRYPT_MODE,currentKeyPack.getKey(),
                      new IvParameterSpec(currentKeyPack.getInitVector()));

      decipherer = Cipher.getInstance(currentKeyPack.getCipherParameters());
      decipherer.init(Cipher.DECRYPT_MODE,currentKeyPack.getKey(),
                      new IvParameterSpec(currentKeyPack.getInitVector()));
    }
    catch(Exception e){
      this.println("**Error in initializing the ciphers**");
    }
  }
  beginPacketTransmission();
  beginPacketReception();
}

/**
 * Provides a byte array containing the key id followed by the
 * given data's ciphertext.
 *
 * @param clearData  the data to be enciphered
 * @return byte[]    an array containing the keyID followed by the ciphertext
 */
private byte[] encipherData(byte[] clearData){
  byte[] cipherArray = null;

  try{

    // synchronized around the currentakeyPack in order to ensure the keyPack does
    //    not change between the .getKeyID call and the encipherer call
    synchronized(currentKeyPack){

      // get the active key ID and encipher the data
      byte[] keyID = currentKeyPack.getKeyID();
      Cipher c = encipherer;
      byte[] encryptedData = c.doFinal(clearData);


      // create the cipherArray containing the key ID in the first 4 bytes,
      //    followed by the encrypted data
      cipherArray = new byte[4 + encryptedData.length];
      int i;
      for (i=0; i<4; i++){cipherArray[i] = keyID[i];}
      for (; i<cipherArray.length; i++){cipherArray[i] = encryptedData[i-4];}
    }
  }
  catch (Exception e){
    e.printStackTrace();
  }
  // return the cipherArray
  return cipherArray;
}
```

```
/**
 * Accepts an array of cipherData and returns the cleartext.
 *
 * @param cipherArray  an array containing an key ID followed by ciphertext
 * @return byte[]      the deciphered data array
 */
private byte[] decipherData(byte[] cipherArray) throws Exception {

  // initialize needed byte arrays
  byte[] keyID = new byte[4];
  byte[] clearData = null;
  byte[] cipherText = new byte[cipherArray.length - 4];

  // parses out the key ID and the ciphertext
  int i;
  for (i=0; i<4; i++){keyID[i] = cipherArray[i];}
  for (; i<cipherArray.length; i++){cipherText[i-4] = cipherArray[i];}

    // synchronized around the currentKeyPack in order to ensure the keyPack does
    //    not change between the .getKeyID call and the encipherer call.
    synchronized(currentKeyPack){

      // checks for a valid key ID
      for (int j=0; j<4;j++){
        if (keyID[j] != currentKeyPack.getKeyID()[j]) {
          throw new Exception(" SF: invalid Key ID in incomming packet");
        }
      }

      // decrypts the data
      try{
          Cipher c = decipherer;
          clearData = c.doFinal(cipherText);
      }
      catch (Exception e){
        e.printStackTrace();
      }
    }

    // return the clear data
    return clearData;
}

/**
 * Begins the transmission of packets
 */
public void beginPacketTransmission(){
  okToTransmit = true;
  this.println("begin transmit");
}

/**
 * Stops the transmission of packets
 */
public void endPacketTransmission(){
  okToTransmit = false;
  this.println("end transmit");
}

/**
 * Begins the reception of packets
 */
public void beginPacketReception(){
  okToReceive = true;
  this.println("begin receive **");
}
```

```
/**
 * Stops the reception of packets
 */
public void endPacketReception(){
  okToReceive = false;
  this.println("end receive **");
}

/**
 * Sets the applications' ID
 *
 * @param id  the application's id long value
 */
public void setApplicationID(long id){
  applicationID = id;
  this.println("Application ID set: " + applicationID);
}

/**
 * sets the Filter's ID, and
 *
 * @param id  the filter's ID
 */
public void setID(String id){
  filterID = id;
  this.println("filter ID set to: " + filterID);
  this.println("registering with SM");
  securityManager = (SecurityManager) getServiceProvider(SecurityManager.class);
  securityManager.addSecureSubscriber(this);
}

/**
 * gets the filter's ID
 *
 * @return String the filter's ID
 */
public String getID(){
  return filterID;
}

/**
 * returns the current active SecretKeyPack
 *
 * @return the current active SecretKeyPack
 */
public SecretKeyPack getCurrentKeyPack(){
  return currentKeyPack;
}

/**
 * returns a Vector of SecretKeyPacks, the current one and any others in the queue
 *
 * @return Vector  the vector of keyPacks
 */

public Vector getAllKeyPacks() {
  Vector temp = new Vector();
  temp.add(currentKeyPack);
  for(int i=0; i<nextKeyPacks.size(); i++){
    temp.add(nextKeyPacks.elementAt(i));
  }
  return nextKeyPacks;
}
```

```
    /**
     * Synonymous with 'System.out.println', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    public void println(String aLine){

      // call the 'print' method, passing the provided String object
      //   concatenated with a carriage return
      this.print(aLine.concat("\n"));
    }

    /**
     * Synonymous with 'System.out.print', only it produces a class
     * specific header before the passed String for ease of output
     * identification
     *
     * @param aLine   the String to print out
     */
    private void print(String aLine){
      System.out.print("  Secure Filter (" + filterID + "): " + aLine);
    }

    /**
     * Returns the filter's type as identified in the
     *    SecureManagerSubscriber Interface
     *
     * @return int  the filter's type
     */
    public int getFilterType(){
      return this.SECURE_FILTER_TYPE;
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[Abdalla00] Abdalla, M., Shavitt, Y., and Wool, A. Key Management for Restricted Multicast Using Broadcast Encryption. *IEEE/ACM Trans. on Networking* 8, 4 (Aug. 2000) 443-454.

[Albert94] Albert, D. (2000, April 6) *Issues in Muse Security*. Retrieved September 25, 2002, from: ftp://sunsite.unc.edu/pub/academic/communications/papers/muds/muse/Security.TXT.

[Benedens99] Benedens, O. Geometry-based Watermarking of 3D Models. *IEEE Computer Graphics and Applications*, 19,1 (Jan./Feb. 1999), 46-55.

[Berghel97] Berghel, H. Watermarking Cyberspace. *Comm. ACM*, 40,11 (Nov. 1997), 19-24.

[Black00] Black, U. *QOS in Wide Area Networks*. Prentice Hall PTR, upper Saddle River, New Jersey, 2000.

[Bullock99] Bullock, A. and Benford, S. An Access Control Framework for Multi-user Collaborative Environments. In *Proc. Int. SIGGROUP Conf. Supporting Group Work*, ACM (Phoenix, Ariz., Nov. 1999), 140-149.

[Capps00] Capps, M., McGregor, D., Brutzman, D., and Zyda, M. NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments. *IEEE Computer Graphics and Applications* 20, 5 (Oct. 2000), 12-15.

[Capps97] Capps, M. and Stotts, D. Research Issues in Developing Networked Virtual Realities: Working Group Report on Distributed System Aspects of Sharing a Virtual Reality. In *Proc. Sixth Workshop Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE (Cambridge, Mass., June 1997), 205-211.

[Cheshire96] Cheshire, S. It's the latency, stupid. May 1996. Available form : http//rescomp.stanford.edu/~cheshire/rants/Latency.html

[Comer00] Comer, D. *Internetworking with TCP/IP*, 4th ed. Prentice Hall Inc., upper Saddle River, New Jersey, 2000.

[Curti] Curtis, P. MUDs Grow Up: Social Virtual Reality in the Real World, in *Compcon Spring '94, Digest of Papers*, IEEE (San Francisco, California, Feb. 1994), 193-200 .

[DOD96] Department of Defense Directive S-3600.1, 1996, cited in Joint Chiefs of Staff, Information Assurance: Legal, Regulatory, Policy and Organization Considerations, Third ed., U.S. Department of Defense, September 17, 1997.

[FIPS198] Federal Information processing Standards Publication 198, *The Keyed-Hash Message Authentication Code*, National Institute of Standards and Technology. March 2002. Retrieved September 25, 2002, from: http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf.

[Flanagan99] Flanagan, D. *Java in a Nutshell*, 3rd ed. O'Reilly and Associates, Inc., Sebastopol, CA, 1999.

[Forrest97] Forrest, S., Hofmeyr, S., and Somayaji, A. Computer Immunology. *Comm. ACM* 40, 3 (Oct. 97), 88-96.

[Foster98a] Foster, I., Karonis, T., and Kesselman, C. Managing Security in High-Performance Distributed Computations. *Cluster Computing* 1,1 (1998), 95-107.

[Foster98b] Foster, I., Kesselman, C., Tsudik, G., and Tuecke, S. A Security Architecture for Computational Grids. *In Proc. 5th Conf. on Computer and Communications Security.* ACM (San Francisco, Calif., Nov. 1998), 83-92.

[Foster01] Foster, I., Kesselman, C., and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications* 15,3 (Fall 2001), 200-222.

[Grabner01] Grabner, M. Smooth High-Quality interactive Visualization. In *Procs. of Spring Conference on Computer Graphic*, IEEE (Budmerice, Slovakia, Apr. 2001), 87-94.

[Grand98] Grand, Mark. *Patterns in Java, Volume 1.* John Wiley & Sons, Inc., New York, 1998.

[Hunter02] Hunter, D., Cagle, K., Dix, C., Kovack, R., Pinnock, J., and Rafter, J. *Beginning XML,* 2nd ed. Wrox Press Ltd., Birmingham, United Kingdom, 2002.

[Jayaram97] Jayaram, N. D. and Morse, P. L. R. Network Security: A Taxonomic View. In *European Conf. Security and Detection,* IEEE (London, Apr. 1997), 124-127.

[Kapolka02] Kapolka, A., McGregor, D., and Capps, M. A Unified Component Framework for Dynamically Extensible Virtual Environments. In *Proc. Fourth International Conf. on Collaborative Virtual Environments,* ACM (Bonn, Germany, Sept. 2002).

[Landwehr94] Landwehr, C. E., Bull A., McDermott, J., and Choi, W. A Taxonomy of Computer Program Security Flaws. *ACM Comput. Surveys* 26, 3 (Sept. 1994), 211-254.

[Liu01] Liu, S. and Silverman, M. A Practical Guide to Biometric Security Technology. *IT Professional* 3,1 (Jan./Feb. 2001), 27-32.

[Macedonia97] Macedonia, M. R. and Zyda, M. J. A Taxonomy for Networked Virtual Environments. *IEEE Multimedia* 4, 2 (Jan./Mar. 1997), 48-56.

[McGregor01] McGregor, D., and Kapolka, A. *NPSNET-V: An Architecture for Creating Scalable Dynamically Extensible Networked Virtual Environments.* Unpublished presentation given at the 2001 MOVES Institute open House. Naval Postgraduate School, Monterey, California, Aug. 2001.

[Michael02] Michael, J., Auguston, M., Roew, N., and Diehle, R. Software Decoys: Intrusion Detection and CounterMeasures. In *Proc. IEEE Workshop on Information Assurance.* IEEE (West Point, New York, June 2002), 130-138.

[Molva00] Molva, R., and Pannetrat, A. Scalable Multicast Security with Dynamic Recipient Groups. *ACM Trans. On Info. And System Security* 3, 3 (Aug. 2000) 136-160.

[NSTISSC00] NSTISSI Document No. 4009, *National Information Systems Security (INFOSEC) Glossary*, National Security Telecommunications and Information Systems Security Committee, September 2000. Retrieved September 25, 2002, from: www.nstissc.gov/Assets/pdf/4009.pdf.

[NTP01] Mills, D. (1999, November 9) *Network Time Protocol General Overview*. Retrieved September 25, 2002, from The Network Time Synchronization Project website of the University of Delaware at: http://www.eecis.udel.edu/~ntp/.

[Oaks01] Oaks, S. *Java Security, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, 2001.

[Pettifer01] Pettifer, S. and Marsh, J. Collaborative Access Model for Shared Virtual Environments. In *Proc. Tenth Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE (Cambridge, Mass, June 2001), 257-262.

[Pritchard00] Pritchard, M. How to Hurt the Hackers: The Inside Scoop on Internet Cheating and How You Can Combat It. *Game Developer* 7, 6 (June 2000), 28-40.

[Salles02] Salles, E. J., Michael, J. B., Capps, M., McGregor, D., and Kapolka, A. Security of Runtime Extensible Virtual Environments. In *Proc. Fourth International Conf. On Collaborative Virtual Environments*, ACM (Bonn, Germany, Sept. 2002).

[Simmons79] Simmons, G. J., Symmetric and Asymmetric Encryption. *ACM Comp. Surveys* 11, 4 (Dec. 1979), 305-330.

[Singhal99] Singhal, S., and Zyda, M. *Networked Virtual Environments: Design and Implementation*. ACM Press-SIGGRAPH Series, New York, 1999.

[Smith00] Smith, M. *Object Signing in Bamboo.* Master's Thesis at the Naval Postgraduate School, (March 2000) Monterey, California.

[Stallings96] Stallings, W. *Data and Computer Communications,* 5th ed. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Stallings99] Stallings, W. *Cryptography and Network Security: Principles and Practice,* 2nd ed. Prentice Hall, Upper Saddle River, New Jersey, 1999.

[Stillerman99] Stillerman, M., Marceau, C., and Stillman, M. Intrusion Detection for Distributed Applications. *Comm. ACM* 42, 7 (July 99), 62-69.

[Vigna98] Vigna, G., and Kemmerer, R. NetStat: A Network-Based Intrusion Detection Approach. In *Proc. Fourteenth Annual Computer Sec. Applications Conf.*, ACM (Scottsdale, Ariz, Dec. 1998), 25-34.

[Washington01] Washington, D. *Implementation of a Multi-Agent Simulation for the NPSNET-V Virtual Environment Research project.* Master's Thesis at the Naval Postgraduate School, (Sept. 2001) Monterey, California.

[Wathen01] Wathen, M. *Dynamic network Area of Interest Management for Virtual Worlds.* Master's Thesis at the Naval Postgraduate School, (Sept. 2001) Monterey, California.

[Yerry84] Yerry, M., and Shepard, M. Automatic three-dimensional mesh generation by the modified-octree technique. Int. J. for Num. Methods in Eng., 20:1965-1990, 1984.

[Younglove01] Younglove, R. Public Key Infrastructure: How It Works. *Computing and Control Engineering*, IEEE 12, 1 (April 2001), 99-102.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Associate Professor James Bret Michael
   Code CS
   Naval Postgraduate School
   Monterey, CA

4. Research Assistant Professor Michael Capps
   Code CS/CM
   Naval Postgraduate School
   Monterey, CA

5. Research Assistant Don McGregor
   Code CS
   Naval Postgraduate School
   Monterey, CA

6. Ernesto Salles
   Code MOVES
   Naval Postgraduate School
   Monterey, CA